USENIX Proceedings of the 10th USENIX Security Symposium *Washington, D.C., USA, August 2001*

conference proceedings

10th USENIX Security Symposium

Washington, D.C., USA
August 13–17, 2001

Sponsored by
The USENIX Association

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Past USENIX Security Proceedings

Security IX	August 2000	Denver, Colorado, USA	$27/35
Security VIII	August 1999	Washington, D.C., USA	$27/35
Security VII	January 1998	San Antonio, Texas, USA	$27/35
Security VI	July 1996	San Jose, California, USA	$27/35
Security V	June 1995	Salt Lake City, Utah, USA	$27/35
Security IV	October 1993	Santa Clara, California, USA	$15/20
Security III	September 1992	Baltimore, Maryland, USA	$30/39
Security II	August 1990	Portland, Oregon, USA	$13/16
Security	August 1988	Portland, Oregon, USA	$7/7

ISBN 1-880446-07-3

Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste.

USENIX Association

Proceedings of the 10th USENIX Security Symposium

August 13–17, 2001
Washington, D.C., USA

Symposium Organizers

Program Chair

Dan Wallach, *Rice University*

Invited Talks Coordinator

Greg Rose, *Qualcomm*

Program Committee

Dirk Balfanz, *Xerox PARC*
Steve Bellovin, *AT&T Labs–Research*
Carl Ellison, *Intel Corporation*
Ian Goldberg, *Zero Knowledge Systems*
Peter Gutmann, *University of Auckland*
Trent Jaeger, *IBM T.J. Watson Research Center*
Teresa Lunt, *Xerox PARC*
Patrick McDaniel, *University of Michigan*
Mudge, *@stake Inc.*
Vern Paxson, *ACIRI*
Avi Rubin, *AT&T Labs–Research*
Fred Schneider, *Cornell University*
Jonathan Trostle, *Cisco*
Wietse Venema, *IBM T.J. Watson Research Center*
David Wagner, *University of California, Berkeley*

The USENIX Association Staff

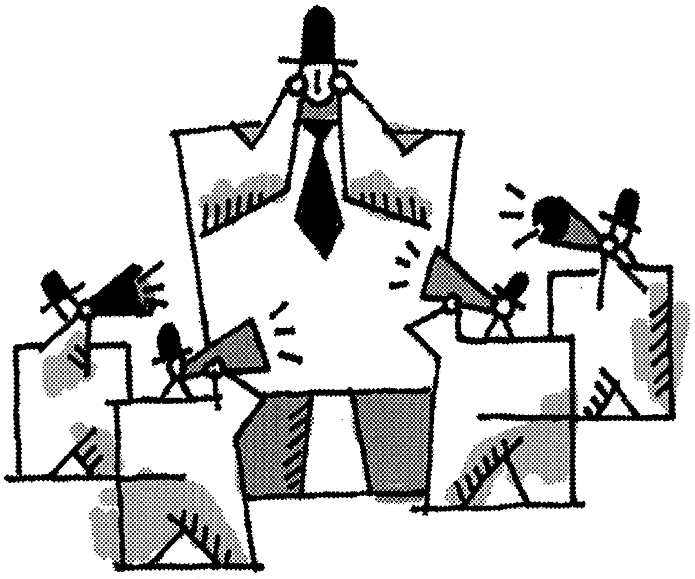
External Reviewers

Tom Berson
Matt Blaze
Drew Dean
Steven Kent
Himanshu Khurana
Andy Lentvorski
Stuart Staniford
Adam Stubblefield
Tara Whalen

10th USENIX Security Symposium

August 13–17, 2001
Washington, D.C., USA

DENIAL OF SERVICE



Session Chair: Steve Bellovin, *AT&T Labs–Research*

Using Client Puzzles to Protect TLS

Drew Dean*
Xerox PARC
ddean@parc.xerox.com
Adam Stubblefield†
Rice University
astubble@rice.edu

Abstract

Client puzzles are commonly proposed as a solution to denial-of-service attacks. However, very few implementations of the idea actually exist, and there are a number of subtle details in the implementation. In this paper, we describe our implementation of a simple and backwards compatible client puzzle extension to TLS. We also present measurements of CPU load and latency when our modified library is used to protect a secure webserver. These measurements show that client puzzles are a viable method for protecting SSL servers from SSL based denial-of-service attacks.

1 Introduction

Denial-of-service attacks have become a major problem on the Internet. Major web sites have been taken down for several hours at a time by distributed denial-of-service (DDoS). The attackers have shown an interesting combination of skill and ignorance. They are able to break into tens or hundreds of machines and install their tool of choice. They then use these "zombie" machines to actually launch the DDoS attack. Some of the tools even use encrypted communications between the attacker and zombie machines. The tools forge the source IP address on the traffic they generate in order to make determining the zombie machine somewhat harder. They will pick IP addresses that are on the same subnet, in order to overcome egress filtering.

However, the tools work via brute force: they just generate random traffic (perhaps with a political message) aimed at a particular machine. While generating a gigabyte per second of traffic aimed at a single machine will bring most websites down to their knees, the sheer volume traffic stands out for anyone doing network monitoring. For ecommerce sites, the attacker could easily arrange an attack such that the website remained available, but web surfers are unable to complete any purchases. Such an attack is based on going after the secure server that processes credit card payments[1]. The SSL/TLS protocol, as it stands, allows the client to request the server to perform an RSA decryption without first having done any work. RSA decryption is an expensive operation; the largest secure site we are aware of can process 4000 RSA decryptions per second. If we assume that a partial SSL handshake takes 200 bytes, then 800 KB/s is sufficient to paralyze an ecommerce site. Such a small amount of traffic is much easier to hide.

The rest of this paper describes our design and implementation of a modification to the TLS protocol to overcome this attack. We use the idea of client puzzles to slow down the attacker sufficiently that a denial-of-service is no longer possible. The rest of the paper is organized as follows: Section 2 discusses related work, Section 3 presents our approach, Section 4 contains an analysis of our proposed scheme, Section 5 discusses future work, and Section 6 concludes.

2 Related Work

The original idea of cryptographic puzzles is due to Merkle [8]. However, Merkle used puzzles for key agreement, rather than access control. Client puzzles have been applied to TCP SYN flooding by Juels and

*This work was supported in part by DARPA grant N66001-00-1-8921.

†This work was completed while the author was an intern at Xerox PARC.

[1]Most ecommerce sites only use a secure server for transmitting payment information.

Brainard [7], who mention that SSL has a similar problem. Aura, Nikander, and Leiwo apply client puzzles to authentication protocols in general [2]; we share a number of techniques with them. However, we concentrate specifically on TLS. Dwork and Naor presented client puzzles as a general solution to controlling resource usage, and specifically for regulating junk email [3]. Their schemes develop along a different axis, primarily motivated by the desire for the puzzles to have shortcuts if a piece of secret information is known. Franklin and Malkhi use iterated application of a strong hash function as a forgery-resistant timer for web usage monitoring [6]. Again, this is slightly different, as we are not interested in counting time, rather in providing a rate limiting step for new TLS connections. Rivest, Shamir, and Wagner posed the related problem of time-lock cryptography in their 1996 paper [9]. Our goal is much much more limited than theirs; we seek only to prevent a denial of service attack on TLS. This implies that we need not concern ourselves with making our puzzles inherently sequential; an attacker capable of solving puzzles in parallel could just as easily launch multiple attacks in parallel.

3 Design and Implementation

It is always critical to be explicit about the goals and assumptions of security work. Our goals are as follows:

1. To prevent denial of service attacks on secure web servers.
2. To remain as backwards compatible as possible with TLS.
3. To minimize additional server load added by implementing these measures.

These goals are listed in order of importance. Clearly, solving the problem we are considering is the first priority. Remaining compatible with existing TLS implementations is much more important than minimizing impact on server performance, because purchasing small additional amounts of CPU power is generally inexpensive.

We make the following assumptions about the environment:

1. That the server being protected has ample network bandwidth. We cannot prevent the brute force DDoS attack; we only aim to prevent an attack against TLS.
2. That legitimate clients, seeking access to a heavily loaded server, are willing to perform a computation that takes no more than a few seconds, and often much less.

These assumptions acknowledge fundamental tradeoffs in availability. We are working at the TLS layer, on top of TCP. If there is insufficient bandwidth for TCP to operate, there is nothing we can do about it here. We require each client to make a small sacrifice in peak performance to make average performance better. This is almost always a good tradeoff to make.

3.1 Design

The TLS protocol breaks up the underlying TCP stream into a record oriented protocol. The unshaded portions of Figure 1 diagram the opening TLS handshake. The TLS specification specifies that unknown (to a particular implementation) record type shall be ignored. Therefore, we use a new record type for the puzzle messages. This allows us to we remain backwards compatible with old TLS implementations that do not support puzzles. Though such implementations may time out a connection if they do not reply to a puzzle, they will not notice any protocol violations. This technique is only applicable to TLS and does not work for SSLv3 as SSLv3 does not discard unknown record types. When the server is not under attack, no changes in the TLS protocol are required.

In order to prevent the denial-of-service attack against TLS, we need to add a new message after the Server Hello message and before the Server Done message. See the shaded portions of Figure 1. This message contains a cryptographic puzzle and is only sent when the server is under load. The server will then wait on a response message before continuing with the handshake protocol.

3.1.1 The Client Puzzles

To be useful as a client puzzle, a puzzle needs to be solvable in a predictable amount of time. The puzzle generally should not take too long to solve (*e.g.*, no more than a second or so on a relatively slow machine), but at the same time, there should be no known shortcut to solving the puzzle. In addition, the server needs to be able

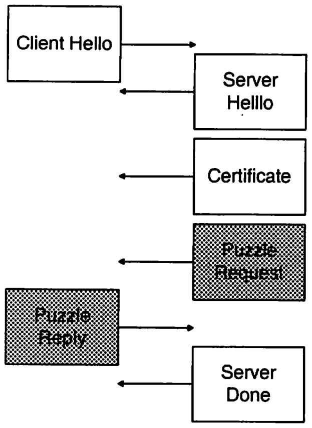


Figure 1: The TLS handshake protocol. The shaded portions are our additions.

```
struct {
 uint8 PuzzleLength;
 uint8 PuzzlePreimage[64];
 uint8 PuzzleHash[16];
} PuzzleChallenge;

struct {
 uint8 PuzzleSolution[64];
} PuzzleResponse;
```

Figure 2: The client puzzle messages

to generate puzzles while doing much less work than the client solving them. Of course, the server also needs an efficient method of verifying the correctness of a proposed solution.

For $h(x)$, a preimage resistant hash function, a client puzzle is the triple $(n, x', h(x))$, where x' is x with its n lowest bits set to 0. Both MD5 and SHA-1 are conjectured to be preimage resistant. The solution to the puzzle is the full value of x. Because $h(x)$ is preimage resistant, the best way for a client to generate x, is to try values in the domain bounded by 0 and x' until a match is found. This should take, on average, 2^{n-1} calculations of $h(x)$. The server, on the other hand, needs to generate a random block (for MD5 and SHA-1, 512 bits) of data, and evaluate the hash function twice. Note that we generate an entire random block rather than just the unknown bits to prevent an attacker from effectively precomputing all possible puzzles.

3.1.2 The Message Format

The client puzzle message format we use is described using the TLS presentation language in Figure 2. The value *PuzzleLength* is the number of unknown bits in *PuzzlePreimage*. *PuzzleHash* is the target value and is computed by taking the MD5 hash of the correct puzzle solution. In the reply message, *PuzzleSolution* is the client's solution to the puzzle. The server checks to make sure that $MD5(PuzzleSolution) = PuzzleHash$.

3.1.3 Determining Server Load

We desire only to send puzzles when the server is overloaded for two reasons:

1. While the server is not overloaded, we would like all TLS clients, not just those with our modifications, to be able to communicate with the server.
2. There is no point to adding more latency to TLS connection setup when the server is not overloaded.

Determining when a server is overloaded due to incoming TLS connections is one of the interesting problems we faced. There is no obviously correct metric: the machine's load is due to many factors, and the rate of new TLS connections may be quite high, but still manageable, *e.g.*, if the machine has a hardware cryptographic accelerator. Note that the computationally intensive section of the TLS handshake protocol occurs after the client sends the *ClientKeyExchange* message[2]. At that point, the server must perform a public key operation. In the most common case, this operation is a RSA private key decryption. This is the operation that we seek to protect the server from having to complete for malicious users.

The metric we choose to use counts the number of RSA decryptions that we have committed to performing. We increment this count when either we decide not to send a client puzzle or when the correct solution to a client

[2]There are other computationally intensive sections of the protocol if the server agrees to an anonymous or export cipher suite. These are beyond the scope of this paper.

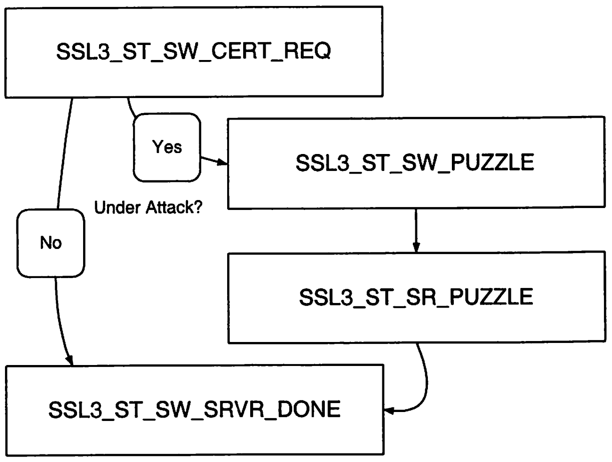


Figure 3: TLS states in OpenSSL

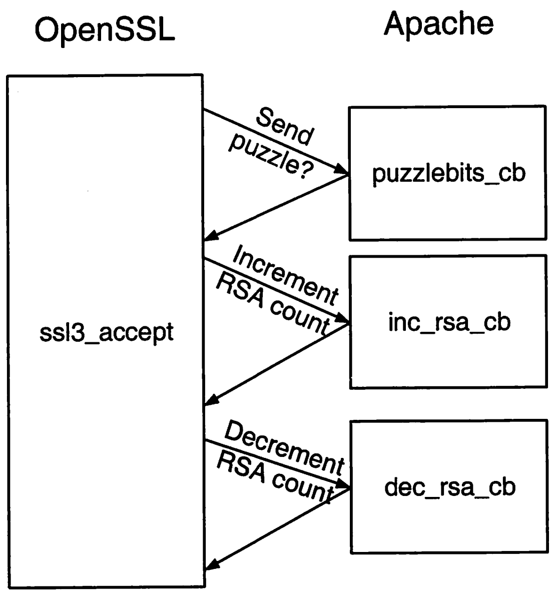


Figure 4: Control flow in OpenSSL with client puzzles

puzzle is submitted. The count is decremented after the corresponding RSA decryption has completed or if the connection is closed before the RSA operation is begun. This measures exactly what we care about: whether there is a backlog of public key operations to be performed in the near future.

By detecting whether this value is above a specified limit, the server can determine whether or not to send a client puzzle. More complicated schemes based on this metric such as a state machine with distinct entrance into and exit from client puzzle mode levels or statistical regressions are also possible.

3.2 Implementation

In order to measure the effectiveness of our solution, we modified the OpenSSL library to support servers and clients that understood our puzzle protocol. On the server side, hooks were added to the mod_ssl Apache module and as a client the TLS enhanced version of lynx was used.

3.2.1 The OpenSSL Library

OpenSSL is an open-source library that includes support for the SSL and TLS protocols as well as the underlying cryptographic operations needed by SSL and TLS [5]. OpenSSL handles connections on a per-socket basis and does not keep any global process state. This prevents a clean separation between our modified OpenSSL library and server applications because we need to measure the global server load. On the client side however, the application never needs to be aware whether the puzzle protocol took place. Clients can trivially support the protocol just by relinking with the modified library.

In OpenSSL, the TLS handshake is implemented as a state machine representing the current location in the protocol. To add support for puzzles on the server, a new state was added after the server certificate request state. In this state the server either sends a puzzle request and switches to a state waiting to receive the puzzle reply or immediately switches to the server done state. The puzzle reply state will wait to receive a puzzle solution before switching to the server done state. If a puzzle solution is never received the connection will time out. This is diagrammed in Figure 3.

On the client side, we treat the reception of a puzzle as an "unexpected event", in the incoming message handler because the client is expecting a handshake record. The puzzle solution is then computed and returned to the server before the handshake processing continues.

The biggest challenge is deciding whether a server should send a client puzzle. Because OpenSSL has no notion of application or system wide state, it has no way to count the number of RSA operations a server has committed to. To remedy this problem, we provide callbacks to alert the application whenever we commit to or finish an RSA private decryption. We also add a callback that

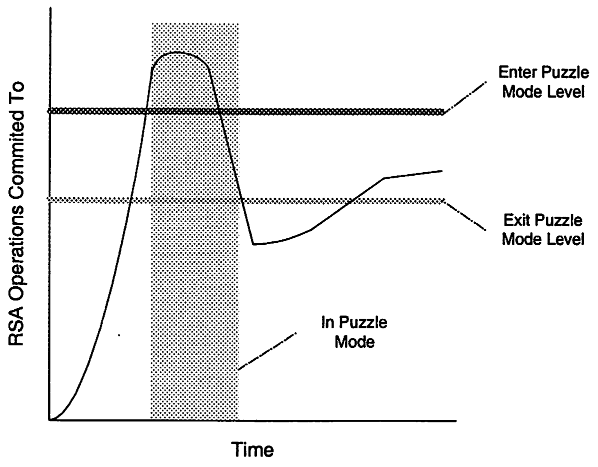


Figure 5: Puzzle states

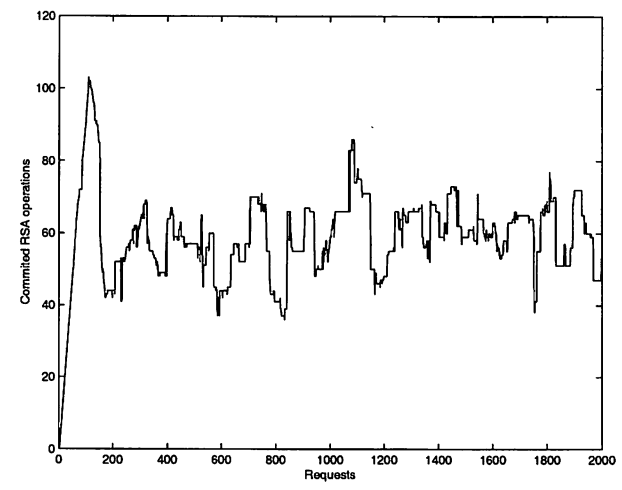


Figure 6: Committed RSA operations at each request without puzzles (during attack)

allows the server to decide whether to send a client puzzle on the current connection, and if so, how many bits the puzzle should be. This control flow is shown in Figure 4.

3.2.2 The Server

Our server implementation was based on the Apache webserver [1] with the mod_ssl module [4]. In Apache, every connection is handled by a different UNIX process. This makes sharing information between connections rather difficult. We needed to count the total number of RSA private key operations that all of the Apache processes had committed to in order to correctly determine when to send client puzzles. We used a page of shared memory, protected by file-based mutexes, to store the count of committed operations, and also whether a puzzle was sent on the last connection. This provided us with a reasonably simple and efficient implementation.

The server uses two user specified values to tell OpenSSL whether to send a client puzzle. One value is the maximum number of private key operations to commit to before sending puzzles. Once it starts sending puzzles it continues until the number of committed operations drops below the other specified value. We use separate thresholds for turning on and off puzzles to provide some hysteresis so that the server does not oscillate in and out of puzzle mode too rapidly. This process is outlined in Figure 5. The selection of these values is described in Section 4.

3.2.3 The Client

Integrating this scheme on the client side was surprisingly simple. It can be added to any client that supports TLS through OpenSSL simply by relinking with the new library. Initially we had planned to add a status message to alert the user when a puzzle was being completed, but the time needed to complete a puzzle is so short that this did not end up being necessary.

4 Analysis

In this section we analyze the effectiveness of our client puzzle protocol in protecting a webserver against a TLS based denial of service attack. To benchmark the server, we used a modified version of Dan Boneh's multithreaded TLS benchmark. Our test server was an 750 MHz Athlon with 256 MB of RAM running FreeBSD 4.0. Using OpenSSL's RSA implementation and benchmarks, the server was able to complete 148 1024-bit private key operations a second.

4.1 Performance Without Client Puzzles

We first ran our benchmark on an copy of Apache version 1.3.12 with mod_ssl version 2.6.5 without support for the puzzle protocol. We did add some minimal profiling support to this build. Using one client and

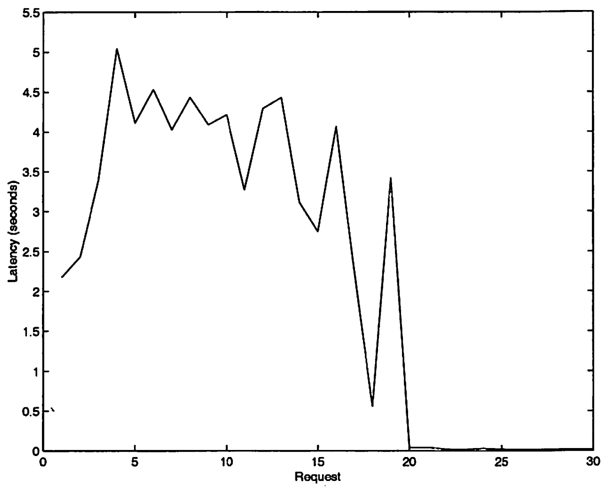


Figure 7: Latency for a legitimate client without puzzles (during and after attack).

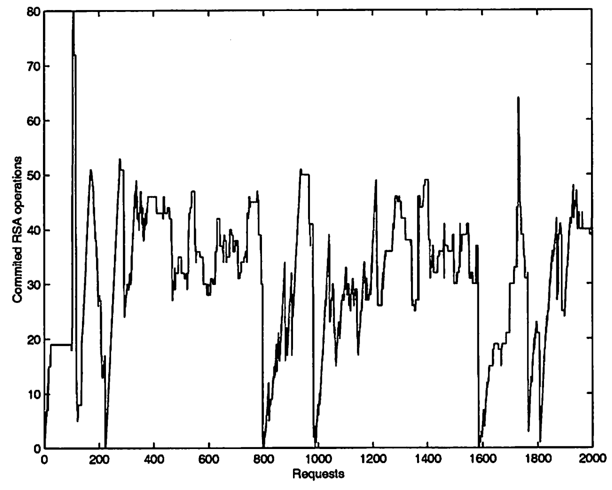


Figure 8: Committed RSA operations at each request with puzzles (during attack). Note the different scale from figure 6.

less than 550Kbps of traffic, we were able to completely load the server. As shown in Figure 6, the number of pending RSA operations was continually increasing for the first 100 TLS connections made to the server during a simulated attack. At this point, there were no more Apache processes available to handle additional requests, so the number of pending requests falls as RSA operations complete with no new operations being committed to, as clients are unable to make new connections to the server. Figure 7 shows the latency experienced by a legitimate user trying to connect to the server during this period during a representative benchmarked run. By using two attacking computers, we were able to double the latency experienced by the legitimate user. These simulated attacks can be continued indefinitely by the attacking computers. These results show that an unprotected TLS server is indeed vulnerable to these attacks.

4.2 Performance With Client Puzzles

The situation when using the client puzzle enabled version of Apache is much better. During the non-client puzzle tests, we observed that the loaded server was able to complete approximately 60 requests per second. We therefore decided to set the upper bounds on committed RSA operations to 40. That way, a client would have to wait no more that a second to connect. After that bound was reached, we decided to leave the system in puzzle mode until the number of committed RSA operations dropped below 10. This prevented the server from switching into and out of puzzle mode too often during a continuous attack. After testing different values for the length of the client puzzles to send, we settled on 20-bit puzzles, as a value that stopped even multiple attackers but did not disrupt client operations. A possible improvement to this scheme would be to increase the puzzle length depending on the length of time that an attack has been taking place. This would slow down attackers even further, but would also cause legitimate client wait times to increase.

Figure 9 shows the latency for a legitimate client during a denial of service attack waged against the modified server using multiple attackers. The number of pending RSA operations is shown in Figure 8. The server console remained usable throughout the attacks and its processor was never completely loaded. The extremely low, and neat constant, latency is the key metric. These results indicate that the puzzle protocol was effective in preventing the TLS based denial of service attack.

4.3 Security Considerations

Because the whole point of using TLS is to provide a secure connection between hosts, we look now to the security implications of our client puzzle protocol. We begin by noting that there are no shared keys between the client puzzle protocol and the handshake protocol. The client puzzle protocol merely "stops" the TLS handshake protocol until it completes. Because the handshake protocol is not time dependent (with the exception of a times-

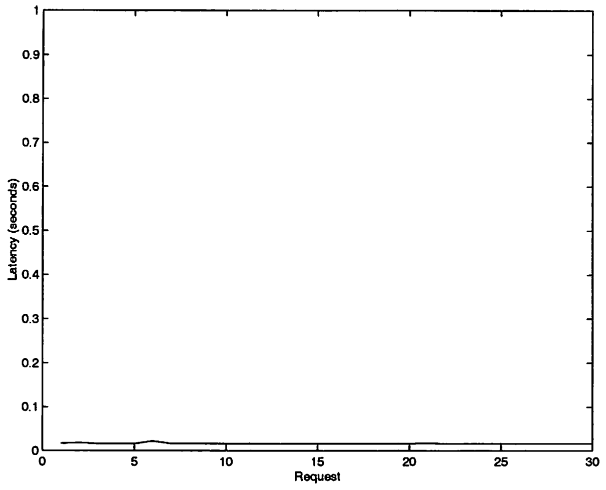


Figure 9: Latency for a legitimate client with puzzles (during and after attack). Note the different scale as compared to Figure 7.

tamp sent at the beginning of the protocol that does not even need to be correct), this does not seem to negatively affect security. The only possible problem is that the puzzle protocol is generating random data from the same pool as the handshake protocol. However, as long as the random number generator used in the TLS implementation is not poly-time distinguishable from true randomness, the client puzzle protocol should have no effect on the security of TLS. Of course, a TLS implementation that uses an insecure random number generator has much more serious security problems that are beyond the scope of this work.

5 Future Work

We have built a functional prototype and examined its behavior. While the system behaves well, further improvements could be made with more sophisticated strategies for determining when to start and stop the puzzle requests. A further degree of control is available by dynamically adjusting puzzle length depending on conditions at that moment on the server. Of course, a security proof for this scheme would be desirable.

6 Conclusions

Client puzzles are an effective means of countering a denial-of-service attack against TLS servers. We have presented an implementation that remains fully compatible with existing TLS clients when the server is not under attack. This is the best possible compatibility. We have shown that puzzle sizes can be chosen that keep the server available, even under duress, while adding latency below the humanly perceptible threshold for interactive response.

Acknowledgments

We thank Matt Franklin and Dan Boneh for useful discussion about this work. We thank Diana Smetters for helpful comments that improved the presentation of this paper.

References

[1] The Apache HTTP server project. `http://www.apache.org/httpd.html`.

[2] Tuomas Aura, Pekka Nikander, and Jussipekka Leiwo. Dos-resistant authentication with client puzzles. In *Proceedings of the Cambridge Security Protocols Workshop 2000*, LNCS, Cambridge, UK, April 2000. Springer-Verlag.

[3] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In Ernest F. Brickell, editor, *Proc. CRYPTO 92*, pages 139–147. Springer-Verlag, 1992. Lecture Notes in Computer Science No. 740.

[4] Ralf S. Engelschall. mod_ssl: The Apache interface to OpenSSL. `http://www.modssl.org/`.

[5] Ralf S. Engelschall. Openssl: The open source toolkit for SSL/TLS. `http://www.openssl.org/`.

[6] Matthew K. Franklin and Dahlia Malkhi. Auditable metering with lightweight security. *Journal of Computer Security*, 6(4):237–255, 1998.

[7] Ari Juels and John Brainard. Client puzzles: A cryptographic defense against connection depletion attacks. In S. Kent, editor, *Proceedings of NDSS '99*, pages 151–165, 1999.

[8] R. C. Merkle. Secure communications over insecure channels. *Communications of the ACM*, 21:294–299, April 1978.

[9] Ronald L. Rivest, Adi Shamir, and David A. Wagner. Time-lock puzzles and timed-release cryptography. (Preliminary version posted on the web by Rivest.).

Inferring Internet Denial-of-Service Activity

David Moore
CAIDA
San Diego Supercomputer Center
University of California, San Diego
dmoore@caida.org

Geoffrey M. Voelker and Stefan Savage
Department of Computer Science and Engineering
University of California, San Diego
{voelker,savage}@cs.ucsd.edu

Abstract

In this paper, we seek to answer a simple question: "How prevalent are denial-of-service attacks in the Internet today?". Our motivation is to understand quantitatively the nature of the current threat as well as to enable longer-term analyses of trends and recurring patterns of attacks. We present a new technique, called "backscatter analysis", that provides an estimate of *worldwide* denial-of-service activity. We use this approach on three week-long datasets to assess the number, duration and focus of attacks, and to characterize their behavior. During this period, we observe more than 12,000 attacks against more than 5,000 distinct targets, ranging from well known e-commerce companies such as Amazon and Hotmail to small foreign ISPs and dial-up connections. We believe that our work is the only publically available data quantifying denial-of-service activity in the Internet.

1 Introduction

In February of 2000, a series of massive denial-of-service (DoS) attacks incapacitated several high-visibility Internet e-commerce sites, including Yahoo, Ebay, and E*trade. Next, in January of 2001, Microsoft's name server infrastructure was disabled by a similar assault. Despite attacks on high-profile sites, the majority of attacks are not well publicized. Many other domestic and foreign sites have also been victims, ranging from smaller commercial sites, to educational institutions, public chat servers and government organizations.

While it is clear from these anecdotal reports that denial-of-service attacks continue to be a problem, there is currently not much quantitative data about the prevalence of these attacks nor any representative characterization of their behavior. Unfortunately, there are multiple obstacles hampering the collection of an authoritative denial-of-service traffic dataset. Service providers and content providers consider such data sensitive and private. Even if it were allowed, monitoring traffic at enough sites to obtain a representative measure of Internet-wide attacks presents a significant logistical challenge. Consequently, the only contemporary public data we are aware of is a CSI/FBI survey study [8][1].

We believe that a strong quantitative foundation is necessary both for understanding the nature of today's threat and as a baseline for longer-term comparison and analysis. Our paper seeks to answer the simple question: "How prevalent are denial-of-service attacks in the Internet today?". As a means to this end, we describe a traffic monitoring technique called "backscatter analysis" for estimating the *worldwide* prevalence of denial-of-service attacks. Using backscatter analysis, we observe 12,805 attacks on over 5,000 distinct Internet hosts belonging to more than 2,000 distinct organizations during a three-week period. We further are able to estimate a lower-bound on the intensity of such attacks – some of which are in excess of 600,000 packets-per-second (pps) – and characterize the nature of the sites victimized.

The remainder of this paper is organized as follows: Section 2 describes the underlying mechanisms of denial-of-service attacks, Section 3 describes the backscatter technique, and limitations arising from its assumptions, and Section 4 explains our techniques for classifying attacks from monitored backscatter traffic. In Section 5 we describe our experimental platform, and present our results in Section 6. Finally, in Sections 7 and 8 we cover related work and summarize our find-

[1]The primary result from this report is that 27 percent of security professionals surveyed detected denial-of-service attacks during the year 2000.

ings.

2 Background

Denial-of-service attacks consume the resources of a remote host or network that would otherwise be used for serving legitimate users. There are two principal classes of attacks: *logic* attacks and *flooding* attacks. Attacks in the first class, such as the "Ping-of-Death", exploit existing software flaws to cause remote servers to crash or substantially degrade in performance. Many of these attacks can be prevented by either upgrading faulty software or filtering particular packet sequences, but they remain a serious and ongoing threat. The second class, flooding attacks, overwhelm the victim's CPU, memory, or network resources by sending large numbers of spurious requests. Because there is typically no simple way to distinguish the "good" requests from the "bad", it can be extremely difficult to defend against flooding attacks. For the purposes of this study we will focus solely on flooding attacks.

2.1 Attack types

There are two related consequences to a flooding attack – the network load induced and the impact on the victim's CPU. To load the network, an attacker generally sends small packets as rapidly as possible since most network devices (both routers and NICs) are limited not by bandwidth but by packet processing rate. Therefore, packets-per-second are usually the best measure of network load during an attack.

An attacker often simultaneously attempts to load the victim's CPU by requiring additional processing above and beyond that required to receive a packet. For example, the best known denial-of-service attack is the "SYN flood" [6] which consists of a stream of TCP SYN packets directed to a listening TCP port at the victim. For each such SYN packet received, the host victim must search through existing connections and if no match is found, allocate a new data structure for the connection. Moreover, the number of these data structures may be limited by the victim's operating system. Consequently, without additional protection, even a small SYN flood can overwhelm a remote host. There are many similar attacks that exploit other code vulnerabilities including TCP ACK, NUL, RST and DATA floods, IP fragment floods, ICMP Echo Request floods, DNS Request floods, and so forth.

2.2 Distributed attacks

While a single host can cause significant damage by sending packets at its maximum rate, attackers can (and do) mount more powerful attacks by leveraging the resources of multiple hosts. Typically an attacker compromises a set of Internet hosts (using manual or semi-automated methods) and installs a small attack daemon on each, producing a group of "zombie" hosts. This daemon typically contains both the code for sourcing a variety of attacks and some basic communications infrastructure to allow for remote control. Using variants of this basic architecture an attacker can focus a coordinated attack from thousands of zombies onto a single site.

Packet sent	Response from victim
TCP SYN (to open port)	TCP SYN/ACK
TCP SYN (to closed port)	TCP RST (ACK)
TCP ACK	TCP RST (ACK)
TCP DATA	TCP RST (ACK)
TCP RST	no response
TCP NULL	TCP RST (ACK)
ICMP ECHO Request	ICMP Echo Reply
ICMP TS Request	ICMP TS Reply
UDP pkt (to open port)	protocol dependent
UDP pkt (to closed port)	ICMP Port Unreach
...	...

Table 1: A sample of victim responses to typical attacks.

2.3 IP spoofing

To conceal their location, thereby forestalling an effective response, attackers typically forge, or "spoof", the IP source address of each packet they send. Consequently, the packets appear to the victim to be arriving from one or more third parties. Spoofing can also be used to "reflect" an attack through an innocent third party. While we do not address "reflector attacks" in this paper, we describe them more fully in Section 3.3.

3 Basic methodology

As noted in the previous section, attackers commonly spoof the source IP address field to conceal the location of the attacking host. The key observation behind our technique is that for direct denial-of-service attacks, most programs select source addresses at random for each packet sent. These programs include all of the most popular distributed attacking tools: Shaft, TFN, TFN2k, trinoo, all variants of Stacheldraht, mstream and Trinity). When a spoofed packet arrives at the victim, the victim usually sends what it believes to be an appropriate response to the faked IP address (such as shown in Table 1). Occasionally, an intermediate network device (such as a router, load balancer, or firewall) may issue its own reply to the attack via an ICMP message [21].

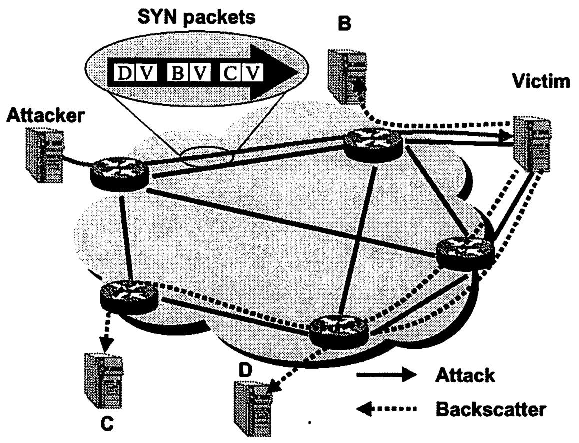


Figure 1: An illustration of backscatter in action. Here the attacker sends a series of SYN packets towards the victim V, using a series of random spoofed source addresses: named C, B, and D. Upon receiving these packets the victim responds by sending SYN/ACKs to each of spoofed hosts.

Again, these ICMP messages are sent to the randomly spoofed source address.

Because the attacker's source address is selected at random, the victim's responses are equi-probably distributed across the entire Internet address space, an inadvertent effect we call "backscatter"[2]. This behavior is illustrated in Figure 1.

3.1 Backscatter analysis

Assuming per-packet random source addresses, reliable delivery and one response generated for every packet in an attack, the probability of a given host on the Internet receiving at least one unsolicited response from the victim is $\frac{m}{2^{32}}$ during an attack of m packets. Similarly, if one monitors n distinct IP addresses, then the expectation of observing an attack is:

$$E(X) = \frac{nm}{2^{32}}$$

By observing a large enough address range we can effectively "sample" all such denial-of-service activity on the Internet. Contained in these samples are the identity of the victim, information about the kind of attack, and a timestamp from which we can estimate attack duration. Moreover, given these assumptions, we can also use the average arrival rate of unsolicited responses directed at the monitored address range to estimate the actual rate of the attack being directed at the victim, as follows:

$$R \geq R' \frac{2^{32}}{n}$$

where R' is the measured average inter-arrival rate of backscatter from the victim and R is the extrapolated attack rate in packets-per-second.

[2]We did not originate this term. It is borrowed from Vern Paxson who independently discovered the same backscatter effect when an attack accidentally disrupted multicast connectivity by selecting global multicast addresses as source addresses [20].

3.2 Address uniformity

The estimation approach outlined above depends on the spoofed source addresses being uniformly distributed across the entire IP address space. To check whether a sample of observed addresses are uniform in our monitored address range, we compute the Anderson-Darling (A2) test statistic [9] to determine if the observations are consistent with a uniform distribution. In particular, we use the implementation of the A2 test as specified in RFC2330 [19] at a 0.05 significance level.

3.3 Analysis limitations

There are three assumptions that underly our analysis:

- *Address uniformity*: attackers spoof source addresses at random.
- *Reliable delivery*: attack traffic is delivered reliably to the victim and backscatter is delivered reliably to the monitor.
- *Backscatter hypothesis*: unsolicited packets observed by the monitor represent backscatter.

We discuss potential biases that arise from these assumptions below.

Key among our assumptions is the random selection of source address. There are three reasons why this assumption may not be valid. First, some ISPs employ *ingress filtering* [12, 5] on their routers to drop packets with source IP addresses outside the range of a customer's network. Thus, an attacker's source address range may not include any of our monitored addresses and we will underestimate the total number of attacks.

"Reflector attacks" pose a second problem for source address uniformity. In this situation, an attacker "launders" the attack by sending a packet spoofed with the victim's source address to a third party. The third party responds by sending a response back towards the victim. If the packets to the third partie are addressed using a broadcast address (as with the popular smurf or fraggle attacks) then third parties may further amplify the attack. The key issue with reflector attacks is that the source address is specifically selected. Unless an IP address in the range we monitor is used as a reflector, we will be unable to observe the attack. We have detected no instances of a monitored host involved in this sort of attack. Our inability to detect, "reflector attacks" cause us to underestimate the total number of denial-of-service attacks.

Finally, if the distribution of source addresses is not random, then any attempt to extrapolate the attack rate via the arrival rate of responses will produce an arbitrarily biased result. This particular problem can be mitigated by verifying that the distribution of observed source addresses is indeed uniform within the set of n addresses we observe.

Another limitation arises from our assumption that packets are delivered reliably and that every packet generates a response. During a large attack it is likely that packets from the attacker may be queued and dropped. Those packets that *do* arrive may be filtered or rate-limited by firewall or intrusion detection software [4] and moreover some forms of attack traffic (e.g., TCP RST messages) do not typically elicit a response. Finally, the responses themselves may be queued and dropped along the path back to our monitored address range. In particular, our estimate of the attack rate is necessarily limited to the capacity of smallest bottleneck link between the victim and our monitor. As with our random distribution assumption, these limitations will cause us to *underestimate* the number of attacks and the attack rate. However, they may also bias our characterization of victims (e.g., if large e-commerce sites are more likely to have rate-limiting software than educational sites, then we may disproportionately underestimate the size of attacks on this class of victim).

The final limitation of our technique is that we assume unsolicited responses represent backscatter from an attack. Any server on the Internet is free to send unsolicited packets to our monitored addresses, and these packets may be misinterpreted as backscatter from an attack. It is possible to eliminate accidental errors by choosing a quiescent address range for monitoring, filtering those packet flows consistently destined to a single host in the range and by high-pass filtering to only record sufficiently long and voluminous packet flows. However, a concerted effort by a third-party to bias our results would be difficult to detect and correct automatically. The most likely source of such bias arises from misinterpretation of random port scans as backscatter. While it is impossible to eliminate this possibility in general, we will show that it is extremely unlikely to be a factor in the vast majority of attacks we observe.

In spite of its limitations, we believe our overall approach is sound and provides at worst a conservative estimate of current denial-of-service activity.

4 Attack Classification

After collecting a large trace of backscatter packets, the first task is post-processing the trace. For this we group collections of related packets into clusters representing attacks. The choice of a specific aggregation methodology presents significant challenges. For example, it is often unclear whether contemporaneous backscatter indicating both TCP and ICMP-based attacks should be classified as a single attack or multiple attacks. More difficult still is the problem of determining the start and end times of an attack. In the presence of significant variability, too lenient a threshold can bias the analysis towards fewer attacks of longer duration and low average packet rates, while too strict an interpretation suggests a large number of short attacks with highly variable rates.

Without knowledge of the intent of the attacker or direct observation of the attack as it orchestrated by the attacker, it is impossible to create a synthetic classification system that will group all types of attacks appropriately for all metrics. Instead, we have chosen to employ two distinct classification methods: a flow-based analysis for classifying individual attacks – how many, how long and what kind – and an event-based method for analyzing the severity of attacks on short time scales.

4.1 Flow-based classification

For the purpose of this study, we define a flow as a series of consecutive packets sharing the same target IP address and IP protocol. We explored several approaches for defining flow lifetimes and settled on a fixed timeout approach: the first packet seen for a target creates a new flow and any additional packets from that target are counted as belonging to that flow if the packets are received within five minutes of the most recent packet in this flow. The choice of parameters here can influence the final results, since a more conservative timeout will tend to suggest fewer, longer attacks, while a shorter timeout will suggest a large number of short attacks. We chose five minutes as a human-sensible balance that is not unduly affected by punctuated attacks or temporary outages.

To reduce noise and traffic generated due to random Internet misconfiguration (for instance, one NetBIOS implementation/configuration sends small numbers unsolicited packets to our monitored address range) we discard all flows that do not have at least 100 packets and a flow duration of at least 60 seconds. These parameters are also somewhat arbitrary, but we believe they represent a reasonable baseline – below such thresholds it seems unlikely that an attack would cause significant damage. Finally, flows must contain packets sent to more than one of our monitored addresses.

We examine each individual flow and extract the following information:

- *TCP flag settings*: whether the flow consists of SYN/ACKs, RSTs, etc.
- *ICMP payload*: for ICMP packets that contain copies of the original packet (e.g. TTL expired) we break out the enclosed addresses, protocols, ports, etc.
- *Address uniformity*: whether the distribution of source addresses within our monitored range passes the Anderson-Darling (A2) test for uniformity to the 0.05 significance level.
- *Port settings*: for source and destination ports (for both UDP and TCP) we record whether the port range is fixed, is uniform under the A2 test, or is non-fixed and non-uniform.
- *DNS information*: the full DNS address of the source address – the victim.
- *Routing information*: the prefix, mask and origin AS as registered in our local BGP table on the morning of February 7th.

We generate a database in which each record characterizes the properties of a single attack.

4.2 Event-based classification

Because the choice of flow parameters can impact the estimated duration of an attack, the flow-based method may obscure interesting time-domain characteristics. In particular, attacks can be highly variable – with periodic bursts of activity – causing the flow-based method to vastly underestimate the short-term impact of an attack and overestimate the long-term impact.

We use an event-based classification method keyed entirely on the victim's IP address over fixed time-windows for examining time-domain qualities, such as the number of simultaneous attacks or the distribution of attack rates, For these analyses we divide our trace into one minute periods and record each *attack event* during this period. An attack event is defined by a victim emitting at least ten backscatter packets during a one minute period. We do not further classify attacks according to protocol type, port, etc, as the goal is to estimate the instantaneous impact on a particular victim. The result of this classification is a database in which each record characterizes the number of victims and the intensity of the attacks in each one minute period.

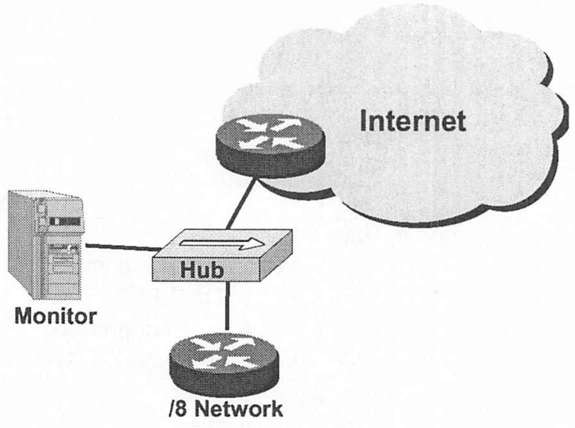


Figure 2: Our experimental backscatter collection platform. We monitor all traffic to our /8 network by passively monitoring data as it is forwarded through a shared hub. This monitoring point represents the only ingress into the network.

5 Experimental platform

For our experiments monitored the sole ingress link into a lightly utilized /8 network (comprising 2^{24} distinct IP addresses, or 1/256 of the total Internet address space). Our monitoring infrastructure, shown in Figure 2, consisted of a PC configured to capture all Ethernet traffic, attached to a shared hub at the router terminating this network. During this time, the upstream router did filter some traffic destined to the network (notably external SNMP queries) but we do not believe that this significantly impacted our results. We also have some evidence that small portions of our address prefix are occasionally "hijacked" by inadvertent route advertisements elsewhere in the Internet, but at worst this should cause us to slightly underestimate attack intensities. We collected three traces, each roughly spanning one week, starting on February 1st and extending to February 25th, and isolated the inbound portion of the network.

6 Results

Using the previously described flows-based approach (Section 4.1), we observed 12,805 attacks over the course of a week. Table 2 summarizes this data, showing more than 5,000 distinct victim IP addresses in more than 2,000 distinct DNS domains. Across the entire period we observed almost 200 million backscatter packets (again, representing less than $\frac{1}{256}$ of the actual attack traffic during this period).

In this section, we first show the overall frequency of attacks seen in our trace, and then characterize the attacks according to both the type of attack and the type of victim.

	Trace-1	Trace-2	Trace-3
Dates (2001)	Feb 01 – 08	Feb 11 – 18	Feb 18 – 25
Duration	7.5 days	6.2 days	7.1 days
Flow-based Attacks:			
Unique victim IPs	1,942	1,821	2,385
Unique victim DNS domains	750	693	876
Unique victim DNS TLDs	60	62	71
Unique victim network prefixes	1,132	1,085	1,281
Unique victim Autonomous Systems	585	575	677
Attacks	4,173	3,878	4,754
Total attack packets	50,827,217	78,234,768	62,233,762
Event-based Attacks:			
Unique victim IPs	3,147	3,034	3,849
Unique victim DNS domains	987	925	1,128
Unique victim DNS TLDs	73	71	81
Unique victim network prefixes	1,577	1,511	1,744
Unique victim Autonomous Systems	752	755	874
Attack Events	112,457	102,204	110,025
Total attack packets	51,119,549	78,655,631	62,394,290

Table 2: Summary of backscatter database.

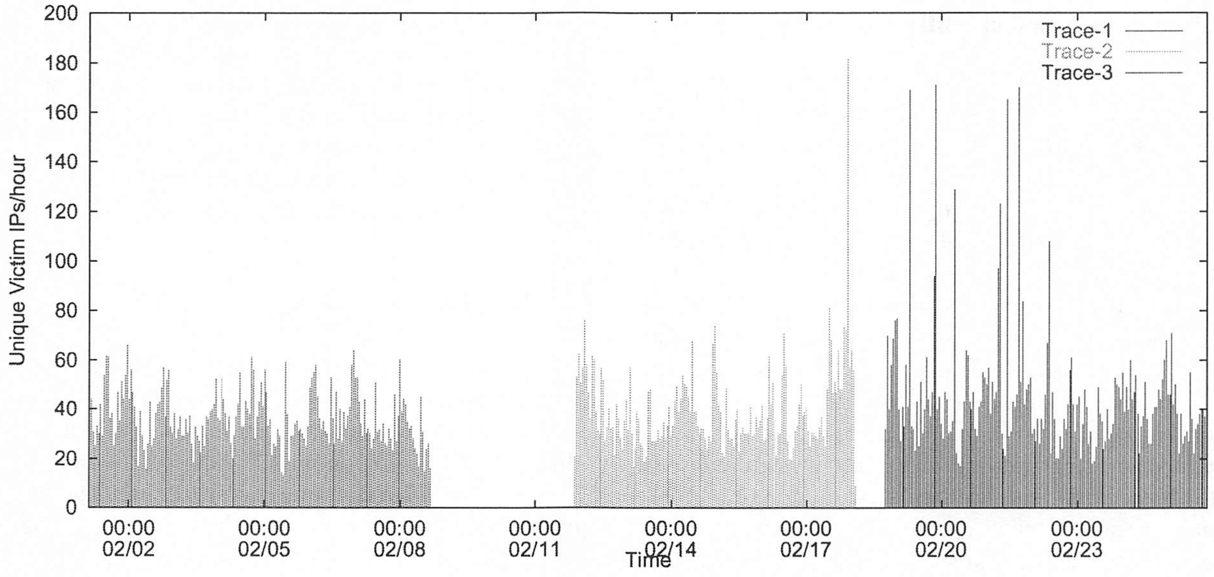


Figure 3: Estimated number of attacks per hour as a function of time (UTC).

Kind	Trace-1		Trace-2		Trace-3	
	Attacks	Packets (k)	Attacks	Packets (k)	Attacks	Packets (k)
TCP (RST ACK)	2,027 (49)	12,656 (25)	1,837 (47)	15,265 (20)	2,118 (45)	11,244 (18)
ICMP (Host Unreachable)	699 (17)	2,892 (5.7)	560 (14)	27,776 (36)	776 (16)	19,719 (32)
ICMP (TTL Exceeded)	453 (11)	31,468 (62)	495 (13)	32,001 (41)	626 (13)	22,150 (36)
ICMP (Other)	486 (12)	580 (1.1)	441 (11)	640 (0.82)	520 (11)	472 (0.76)
TCP (SYN ACK)	378 (9.1)	919 (1.8)	276 (7.1)	1,580 (2.0)	346 (7.3)	937 (1.5)
TCP (RST)	128 (3.1)	2,309 (4.5)	269 (6.9)	974 (1.2)	367 (7.7)	7,712 (12)
TCP (Other)	2 (0.05)	3 (0.01)	0 (0.00)	0 (0.00)	1 (0.02)	0 (0.00)

Table 3: Breakdown of response protocols.

6.1 Time series

Figure 3 shows a time series graph of the estimated number of actively attacked victims throughout the three traces, as sampled in one hour periods. There are two gaps in this graph corresponding to the gaps between traces. In contrast to other workloads, such as HTTP, the number of active attacks does not appear to follow any diurnal pattern (at least as observed from a single location). The outliers on the week of February 20th, with more than 150 victim IP addresses per hour, represent broad attacks against many machines in a common network. While most of the backscatter data averages one victim IP address per network prefix per hour, the ratio climbs to above five for many outliers.

6.2 Attack classification

In this section we characterize attacks according to the protocols used in response packets sent by victims, the protocols used in the original attack packets, and the rate and durations of attacks.

6.2.1 Response protocols

In Table 3 we decompose our backscatter data according to the protocols of responses returned by the victim or an intermediate host. For each trace we list both the number of attacks and the number backscatter packets for the given protocol. The numbers in parentheses show the relative percentage represented by each count. For example, 1,837 attacks in Trace 2 (47% of the total), were derived from TCP backscatter with the RST and ACK flags set.

We observe that over 50% of the attacks and 20% of the backscatter packets are TCP packets with the RST flag set. Referring back to Table 1 we see that RST is sent in response to either a SYN flood directed against a closed port or some other unexpected TCP packet. The next largest protocol category is ICMP host unreachable, comprising roughly 15% of the attacks. Almost all of these ICMP messages contain the TCP header from a packet directed at the victim, suggesting a TCP flood of some sort. Unfortunately, the TCP flags field cannot be recovered, because the ICMP response only includes the first 28 bytes of the original IP packet. ICMP host unreachable is generally returned by a router when a packet cannot be forwarded to its destination. Probing some of these victims we confirmed that a number of them could not be reached, but most were accessible, suggesting intermittent connectivity. This discontinuous reachability is probably caused by explicit "black holing' on the part of an ISP.

We also see a number of SYN/ACK backscatter packets (likely sent directly in response to a SYN flood on an open port) and an equivalent number of assorted ICMP messages, including ICMP echo reply (resulting from ICMP echo request floods), ICMP protocol unreachable (sent in response to attacks using illegal combinations of TCP flags), ICMP fragmentation needed (caused by attacks with the "Dont Fragment" bit set) and ICMP administratively filtered (likely the result of some attack countermeasure). However, a more surprising finding is the large number of ICMP TTL exceeded messages – comprising between 36% and 62% of all backscatter packets observed, yet less than 15% of the total attacks. In fact, the vast majority of these packets occur in just a few attacks, including three attacks on @Home customers, two on China Telecom (one with almost 9 million backscatter packets), and others directed at Romania, Belgium, Switzerland and New Zealand. The attack on the latter was at an extremely high rate, suggesting an attack of more than 150,000 packets per second. We are unable to completely explain the mechanism for the generation of these time-exceeded messages. Upon examination of the encapsulated header that is returned, we note that several of them share identical "signatures" (ICMP Echo with identical sequence number, identification fields, and checksum) suggesting that a single attack tool was in use.

6.2.2 Attack protocols

We refine this data in Table 4 to show the distribution of *attack protocols*. That is, the protocol which must

Kind	Trace-1				Trace-2				Trace-3			
	Attacks		Packets (k)		Attacks		Packets (k)		Attacks		Packets (k)	
TCP	3,902	(94)	28,705	(56)	3,472	(90)	53,999	(69)	4,378	(92)	43,555	(70)
UDP	99	(2.4)	66	(0.13)	194	(5.0)	316	(0.40)	131	(2.8)	91	(0.15)
ICMP	88	(2.1)	22,020	(43)	102	(2.6)	23,875	(31)	107	(2.3)	18,487	(30)
Proto 0	65	(1.6)	25	(0.05)	108	(2.8)	43	(0.06)	104	(2.2)	49	(0.08)
Other	19	(0.46)	12	(0.02)	2	(0.05)	1	(0.00)	34	(0.72)	52	(0.08)

Table 4: Breakdown of protocols used in attacks.

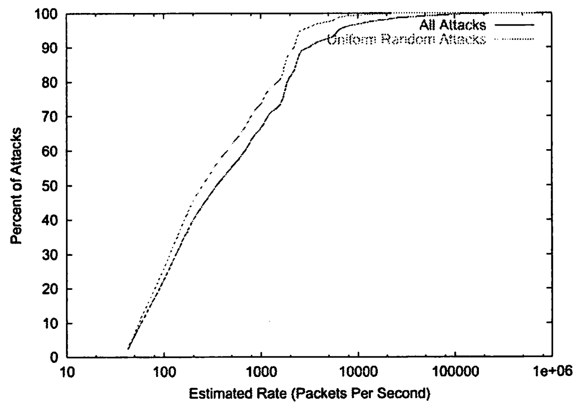


Figure 4: Cumulative distributions of estimated attack rates in packets per second.

have been used by the attacker to produce the backscatter monitored at our network. We see that more than 90% of the attacks use TCP as their protocol of choice, but a smaller number of ICMP-based attacks produce a disproportionate number of the backscatter packets seen. Other protocols represent a minor number of both attacks and backscatter packets. This pattern is consistent across all three traces.

In Table 5 we further break down our dataset based on the service (as revealed in the victim's port number) being attacked. Most of the attacks focus on multiple ports, rather than a single one and most of these are well spread throughout the address range. Many attack programs select random ports above 1024; this may explain why less than 25% of attacks show a completely uniform random port distribution according to the A2 test. Of the remaining attacks, the most popular static categories are port 6667 (IRC), port 80 (HTTP), port 23 (Telnet), port 113 (Authd). The large number of packets directed at port 0 is an artifact of our ICMP categorization – there are fewer than ten TCP attacks directed at port 0, comprising a total of less than 9,000 packets.

6.2.3 Attack rate

Figure 4 shows two cumulative distributions of attack event rates in packets per second. The lower curve shows the cumulative distribution of event rates for all attacks, and the upper curve shows the cumulative distribution of event rates for uniform random attacks, i.e., those attacks whose source IP addresses satisfied the A2 uniform distribution test described in Section 3.2. As described earlier, we calculate the attack event rate by multiplying the average arrival rate of backscatter packets by 256 (assuming that an attack represents a random sampling across the entire address space, of which we monitor $\frac{1}{256}$). Almost all attacks have no dominant mode in the address distribution, but sometimes small deviations from uniformity prevent the A2 test from being satisfied. For this reason we believe that there is likely some validity in the extrapolation applied to the complete attack dataset. Note that the attack rate (x-axis) is shown using a logarithmic scale.

Comparing the distributions, we see that the uniform random attacks have a lower rate than the distribution of all attacks, but track closely. Half of the uniform random attack events have a packet rate greater than 250, whereas half of all attack events have a packet rate greater than 350. The fastest uniform random event is over 517,000 packets per second, whereas the fastest overall event is over 679,000 packets per second.

How threatening are the attacks that we see? Recent experiments with SYN attacks on commercial platforms show that an attack rate of only 500 SYN packets per second is enough to overwhelm a server [10]. In our trace, 38% of uniform random attack events and 46% of all attack events had an estimated rate of 500 packets per second or higher. The same experiments show that even with a specialized firewall designed to resist SYN floods, a server can be disabled by a flood of 14,000 packets per second. In our data, 0.3% of the uniform random attacks and 2.4% of all attack events would still compromise these attack-resistant firewalls. We conclude that the majority of the attacks that we have monitored are fast enough to overwhelm commodity solutions, and a small fraction are fast enough to overwhelm even optimized countermeasures.

Of course, one significant factor in the question of threat posed by an attack is the connectivity of the victim. An attack rate that overwhelms a cable modem victim may be trivial a well-connected major server installation. Victim connectivity is a difficult to ascertain with-

Kind	Trace-1		Trace-2		Trace-3	
	Attacks	Packets (k)	Attacks	Packets (k)	Attacks	Packets (k)
Multiple Ports	2,740 (66)	24,996 (49)	2,546 (66)	45,660 (58)	2,803 (59)	26,202 (42)
Uniformly Random	655 (16)	1,584 (3.1)	721 (19)	5,586 (7.1)	1,076 (23)	15,004 (24)
Other	267 (6.4)	994 (2.0)	204 (5.3)	1,080 (1.4)	266 (5.6)	410 (0.66)
Port Unknown	91 (2.2)	44 (0.09)	114 (2.9)	47 (0.06)	155 (3.3)	150 (0.24)
HTTP (80)	94 (2.3)	334 (0.66)	79 (2.0)	857 (1.1)	175 (3.7)	478 (0.77)
0	78 (1.9)	22,007 (43)	90 (2.3)	23,765 (30)	99 (2.1)	18,227 (29)
IRC (6667)	114 (2.7)	526 (1.0)	39 (1.0)	211 (0.27)	57 (1.2)	1,016 (1.6)
Authd (113)	34 (0.81)	49 (0.10)	52 (1.3)	161 (0.21)	53 (1.1)	533 (0.86)
Telnet (23)	67 (1.6)	252 (0.50)	18 (0.46)	467 (0.60)	27 (0.57)	160 (0.26)
DNS (53)	30 (0.72)	39 (0.08)	3 (0.08)	3 (0.00)	25 (0.53)	38 (0.06)
SSH (22)	3 (0.07)	2 (0.00)	12 (0.31)	397 (0.51)	18 (0.38)	15 (0.02)

Table 5: Breakdown of attacks by victim port number.

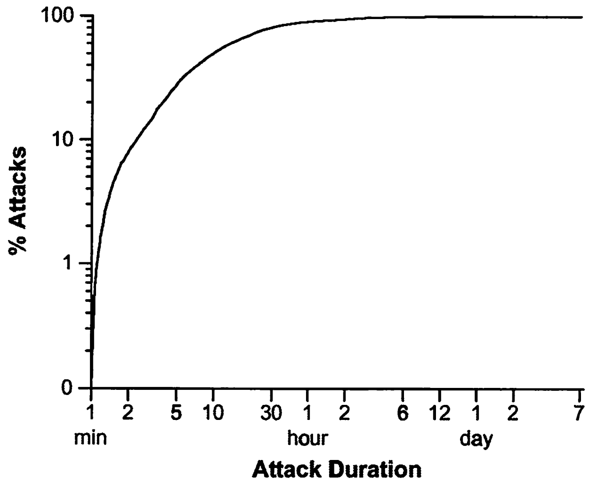


Figure 5: Cumulative distribution of attack durations.

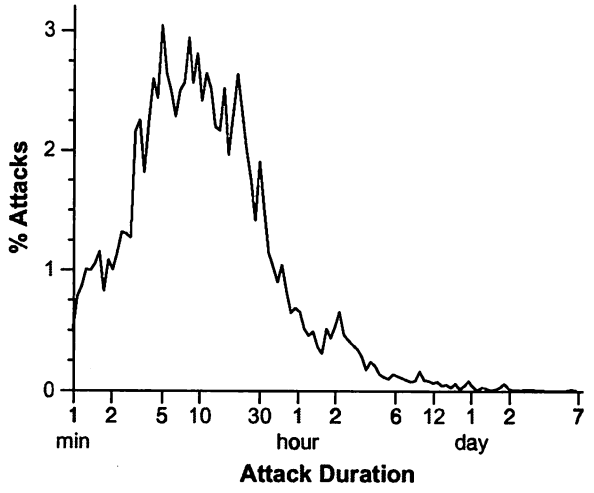


Figure 6: Probability density of attack durations.

out flooding the victim's link. Consequently, we leave correlation between attack rates and victim connectivity as an open problem.

6.2.4 Attack duration

While attack event rates characterize the intensity of attacks, they do not give insight on how long attacks are sustained. For this metric, we characterize the duration of attacks in Figures 5 and 6 across all three weeks of trace data. In these graphs, we use the flow-based classification described in Section 4 because flows better characterize attack durations while remaining insensitive to intensity. We also combine all three weeks of attacks for clarity; the distributions are nearly dentical for each week, and individual weekly curves overlap and obscure each other.

Figure 5 shows the cumulative distribution of attack durations in units of time; note that both the axes are logarithmic scale. In this graph we see that most attacks are relatively short: 50% of attacks are less than 10 minutes in duration, 80% are less than 30 minutes, and 90% last less than an hour. However, the tail of the distribution is long: 2% of attacks are greater than 5 hours, 1% are greater than 10 hours, and dozens spanned multiple days.

Figure 6 shows the probability density of attack durations as defined using a histogram of 150 buckets in the log time domain. The x-axis is in logarithmic units of time, and the y-axis is the percentage of attacks that lasted a given amount of time. For example, when the curve crosses the y-axis, it indicates that approximately 0.5% of attacks had a duration of 1 minute. As we saw in the CDF, the bulk of the attacks are relatively short, lasting from 3–20 minutes. From this graph, though, we see that there are peaks at rounded time durations in this interval at durations of 5, 10, and 20 minutes. Immediately before this interval there is a peak at 3 minutes, and immediately after a peak at 30 minutes. For attacks with longer durations, we see a local peak at 2 hours in the long tail.

6.3 Victim classification

In this section we characterize victims according to DNS name, top-level domain, Autonomous System, and degree of repeated attacks.

6.3.1 Victim Name

Table 6 shows the distribution of attacks according to the DNS name associated with the victim's IP address. We classify these using a hand-tuned set of regular expression matches (i.e. DNS names with "dialup" represent modems, "dsl" or "home.com" represent broadband, etc). The majority of attacks are not classified by this scheme, either because they are not matched by our criteria (shown by "other"), or more likely, because there was no valid reverse DNS mapping (shown by "In-Addr Arpa").

Of the remaining attacks there are several interesting observations. First, there is a significant fraction of attacks directed against home machines – either dialup or broadband. Some of these attacks, particularly those directed towards cable modem users, constitute relatively large, severe attacks with rates in the thousands of packets per second. This suggests that minor denial-of-service attacks are frequently being used to settle personal vendettas. In the same vein we anecdotally observe a significant number of attacks against victims running "Internet Relay Chat" (IRC), victims supporting multi-player game use (e.g. battle.net), and victims with DNS names that are sexually suggestive or incorporate themes of drug use. We further note that many reverse DNS mappings have been clearly been compromised by attackers (e.g., DNS translations such as "is.on.the.net.illegal.ly" and "the.feds.cant.secure.their.shellz.ca").

Second, there is a small but significant fraction of attacks directed against network infrastructure. Between 2–3% of attacks target name servers (e.g., ns4.reliablehosting.com), while 1–3% target routers (e.g., core2-corel-oc48.paol.above.net). Again, some of these attacks, particularly a few destined towards routers, are comprised of a disproportionately large number of packets. This point is particularly disturbing, since overwhelming a router could deny service to *all* end hosts that rely upon that router for connectivity.

Finally, we are surprised at the diversity of different commercial attack targets. While we certainly find attacks on bellwether Internet sites including aol.com, akamai.com, amazon.com and hotmail.com, we also see attacks against a large range of smaller and medium sized businesses.

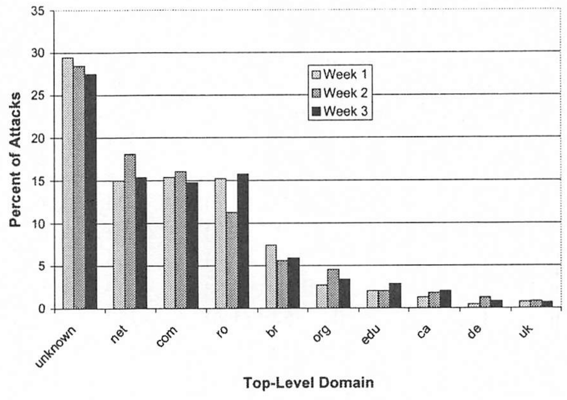


Figure 7: Distribution of attacks to the 10 top-level domains (TLDs) that received the most number of attacks.

6.3.2 Top-level domains

Figure 7 shows the distribution of attacks to the 10 most frequently targeted top-level domains (TLDs). For each TLD displayed on the x-axis, we show one value for each of the three weeks of our study in progressive shades of grey. Note that the TLDs are sorted by overall attacks across all three weeks.

Comparing the number of attacks to each TLD from week to week, we see that there is little variation. Each TLD is targeted by roughly the same percentage of attacks each week. The domain `unknown` represents those attacks in which a reverse DNS lookup failed on the victim IP address (just under 30% of all attacks). In terms of the "three-letter" domains, both `com` and `net` were each targeted by roughly 15% of the attacks, but `edu` and `org` were only targeted by 2–4% of the attacks. This is not surprising, as sites in the `com` and `net` present more attractive and newsworthy targets. Interestingly, although one might have expected attacks to sites in `mil`, `mil` did not show up in any of our reverse DNS lookups. We do not yet know what to conclude from this result; for example, it could be that `mil` targets fall into our `unknown` category.

In terms of the country-code TLDs, we see that there is a disproportionate concentration of attacks to a small group of countries. Surprisingly, Romania (`ro`), a country with a relatively poor networking infrastructure, was targeted nearly as frequently as `net` and `com`, and Brazil (`br`) was targeted almost more than `edu` and `org` combined. Canada, Germany, and the United Kingdom were all were targeted by 1–2% of attacks.

6.3.3 Autonomous Systems

As another aggregation of attack targets, we examined the distribution of attacks to Autonomous Systems (ASes). To determine the origin AS number associated

Kind	Trace-1		Trace-2		Trace-3	
	Attacks	Packets (k)	Attacks	Packets (k)	Attacks	Packets (k)
Other	1,917 (46)	19,118 (38)	1,985 (51)	25,305 (32)	2,308 (49)	17,192 (28)
In-Addr Arpa	1,230 (29)	16,716 (33)	1,105 (28)	24,645 (32)	1,307 (27)	26,880 (43)
Broadband	394 (9.4)	9,869 (19)	275 (7.1)	13,054 (17)	375 (7.9)	8,513 (14)
Dial-Up	239 (5.7)	956 (1.9)	163 (4.2)	343 (0.44)	276 (5.8)	1,018 (1.6)
IRC Server	110 (2.6)	461 (0.91)	88 (2.3)	2,289 (2.9)	111 (2.3)	6,476 (10)
Nameserver	124 (3.0)	453 (0.89)	84 (2.2)	2,796 (3.6)	90 (1.9)	451 (0.72)
Router	58 (1.4)	2,698 (5.3)	76 (2.0)	4,055 (5.2)	125 (2.6)	682 (1.1)
Web Server	54 (1.3)	393 (0.77)	64 (1.7)	5,674 (7.3)	134 (2.8)	730 (1.2)
Mail Server	38 (0.91)	156 (0.31)	35 (0.90)	71 (0.09)	26 (0.55)	292 (0.47)
Firewall	9 (0.22)	7 (0.01)	3 (0.08)	3 (0.00)	2 (0.04)	1 (0.00)

Table 6: Breakdown of victim hostnames.

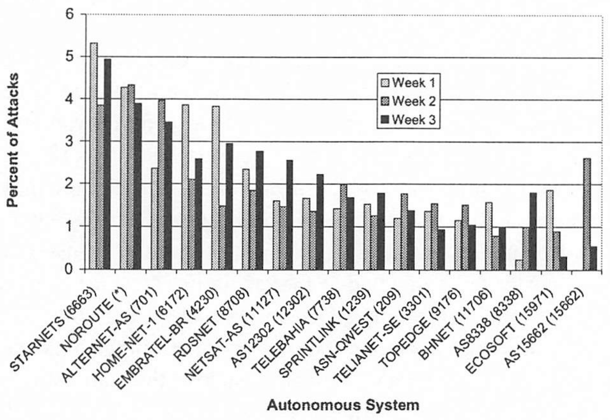


Figure 8: Distribution of attacks to Autonomous Systems (ASes) that were targeted by at least 1% of all attacks.

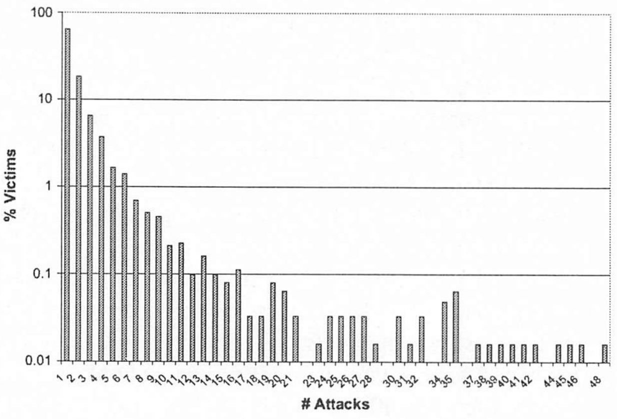


Figure 9: Histogram counting the number of victims of repeated attacks across all traces.

with the victim of an attack, we performed longest prefix matching against a BGP routing table using the victim's IP address. To construct this table, we took a snapshot from a border router with global routes on February 7, 2001. We then mapped AS numbers to identifying names using the NetGeo [17] service to do lookups in registry whois servers. We labeled addresses which had no matching prefix as "NOROUTE".

Figure 8 shows the distribution of attacks to the 17 ASes that were targeted by at least 1% of all attacks. As with top-level domains, each AS named on the x-axis is associated with three values, one for each of the three weeks of our study in progressive shades of grey. Note that the ASes are sorted by overall attacks across all three weeks.

From Figure 8, we see that no single AS or small set of ASes is the target of an overwhelming fraction of attacks: STARNETS was attacked the most, but only received 4-5% of attacks. However, the distribution of ASes attacked does have a long tail. The ASes shown in Figure 8 accounted for 35% of all attacks, yet these ASes correspond to only 3% of all ASes attacked. About 4% of attacks each week had no route according to our offline snapshot of global routes.

Compared with TLDs, ASes experienced more variation in the number of attacks targeted at them for each week. In other words, there is more stability in the type or country of victims than the ASes in which they reside. For example, EMBRATEL's percentage of attacks varies by more than a factor of 2, and AS 15662, an unnamed AS in Yugoslavia, did not show up in week 1 of the traces.

6.3.4 Victims of repeated attacks

Figure 9 shows a histogram of victims of repeated attacks for all traces combined. The values on the x-axis correspond to the number of attacks to the same victim in the trace period, and the values on the y-axis show what percentage of victims were attacked a given number of times in logarithmic scale. For example, the majority of victims (65%) were attacked only once, and many of the remaining victims (18%) were attacked twice. Overall, most victims (95%) were attacked five or fewer times. For the remaining victims, most were attacked less than a dozen times, although a handful of hosts were attacked quite often. In the trace period, one host was attacked 48 times for durations between 72 seconds and 5 hours (at times simultaneously). The graph is also truncated: there are 5 outlier victims attacked 60–70 times, and one unfortunate victim attacked 102 times in a one week span.

6.4 Validation

The backscatter hypothesis states that unsolicited packets represent responses to spoofed attack traffic. This theory, which is at the core of our approach, is difficult to validate beyond all doubt. However, we can increase our confidence significantly through careful examination of the data and via related experiments.

First, an important observation from Table 3 is that roughly 80% of attacks and 98% of packets are attributed to backscatter that does not itself provoke a response (e.g. TCP RST, ICMP Host Unreachable). Consequently, these packets could not have been used for probing our monitored network; therefore network probing is not a good alternative explanation for this traffic.

Next, we were able to duplicate a portion of our analysis using data provided by Vern Paxson taken from several University-related networks in Northern California. This new dataset covers the same period, but only detects TCP backscatter with the SYN and ACK flags set. The address space monitored was also much smaller, consisting of three /16 networks ($\frac{3}{65536}$'s of the total IP address space). For 98% of the victim IP addresses recorded in this smaller dataset, we find a corresponding record at the same time in our larger dataset. We can think of no other mechanism other than backscatter that can explain such a close level of correspondence.

Finally, Asta Networks provided us with data describing denial-of-service attacks directly detected at monitors covering a large backbone network. While their approach and ours capture different sets of attacks (in part due ingress filtering as discussed in Section 3 and in part due to limited peering in the monitored network), their data qualitatively confirms our own; in particular we were able to match several attacks they directly observed with contemporaneous records in our backscatter database.

7 Related work

While denial-of-service has long been recognized as a problem [14, 18], there has been limited research on the topic. Most of the existing work can be roughly categorized as being focused on tolerance, diagnosis and localization. The first category is composed of both approaches for mitigating the impact of specific attacks [4, 16] and general system mechanisms [25, 1] for controlling resource usage on the victim machine. Usually such solutions involve a quick triage on data packets so minimal work is spent on the attacker's requests and the victim can tolerate more potent attacks before failing. These solutions, as embodied in operating systems, firewalls, switches and routers, represent the dominant current industrial solution for addressing denial-of-service attacks.

The second area of research, akin to traditional intrusion detection, is about techniques and algorithms for automatically detecting attacks as they occur [22, 13]. These techniques generally involve monitoring links incident to the victim and analyzing patterns in the arriving and departing traffic to determine if an attack has occurred.

The final category of work, focuses on identifying the source(s) of DoS attacks in the presence of IP spoofing. The best known and most widely deployed of these proposals is so-called *ingress* and *egress* filtering [12, 5]. These techniques, which differ mainly in whether they are manually or automatically configured, cause routers to drop packets with source addresses that are not used by the customer connected to the receiving interface. Given the practical difficulty of ensuring that all networks are filtered, other work has focused on developing tools and mechanisms for tracing flows of packets through the network independent of their ostensibly claimed source address [3, 26, 23, 2, 24, 11].

There is a dearth of research concerned with quantifying attacks within the Internet – denial-of-service or otherwise. Probably the best known prior work is Howard's PhD thesis – a longitudinal study of incident reports received by the Computer Emergency Response Team (CERT) from 1989 to 1995 [15]. Since then, CERT has started a new project, called AIR-CERT, to automate the collection of intrusion detection data from a number of different organizations, but unfortunately their results are not yet available [7]. To our knowledge ours is the only quantitative and empirical study of wide-area denial-of-service attacks to date.

8 Conclusions

In this paper we have presented a new technique, "backscatter analysis," for estimating denial-of-service attack activity in the Internet. Using this technique, we have observed widespread DoS attacks in the Internet, distributed among many different domains and ISPs. The size and length of the attacks we observe are heavy-tailed, with a small number of long attacks constituting a significant fraction of the overall attack volume. Moreover, we see a surprising number of attacks directed at a few foreign countries, at home machines, and towards particular Internet services.

Acknowledgments

We would like to thank a number of people for their contributions to this project. We are particularly grateful to Brian Kantor and Jim Madden of UCSD who provided access to key network resources and helped us understand the local network topology. kc claffy and Colleen Shannon at CAIDA provided support and valuable feedback throughout the project. David Wetherall and Gretta Bartels at Asta Networks donated their time, data and insight. Vern Paxson of ACIRI also provided valuable data and feedback at several stages of our thinking. Finally, we thank the anonymous reviewers for their comments and suggestions. Support for this work was provided by DARPA NGI Contract N66001-98-2-8922, NSF grant NCR-9711092, and Asta Networks.

References

[1] Gaurav Banga, Peter Druschel, and Jeffrey Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proceedings of the 1999 USENIX/ACM Symposium on Operating System Design and Implementation*, pages 45–58, February 1999.

[2] Steven M. Bellovin. ICMP Traceback Messages. Internet Draft: draft-bellovin-itrace-00.txt, March 2000.

[3] Hal Burch and Bill Cheswick. Tracing Anonymous Packets to Their Approximate Source. In *Proceedings of the 2000 USENIX LISA Conference*, pages 319–327, New Orleans, LA, December 2000.

[4] Cisco Systems. Configuring TCP Intercept (Prevent Denial-of-Service Attacks). Cisco IOS Documentation, December 1997.

[5] Cisco Systems. Unicast Reverse Path Forwarding. Cisco IOS Documentation, May 1999.

[6] Computer Emergency Response Team. CERT Advisory CA-1996-21 TCP SYN Flooding Attacks. `http://www.cert.org/advisories/CA-1996-21.html`, September 1996.

[7] Computer Emergency Response Team. AirCERT. `http://www.cert.org/kb/aircert/`, 2000.

[8] Computer Security Institute and Federal Bureau of Investigation. 2000 CSI/FBI Computer Crime and Security Survey. Computer Security Institute publication, March 2000.

[9] R. D'Agostino and M. Stephens. *Goodness-of-Fit Techniques*. Marcel Dekker, Inc., 1986.

[10] Tina Darmohray and Ross Oliver. Hot Spares For DoS Attacks. *;login:*, 25(7), July 2000.

[11] Drew Dean, Matt Franklin, and Adam Stubblefield. An Algebraic Approach to IP Traceback. In *Proceedings of the 2001 Network and Distributed System Security Symposium*, San Diego, CA, February 2001.

[12] P. Ferguson and D. Senie. Network Ingress Filtering: Defeating Denial of Service Attacks Which Employ IP Source Address Spoofing. RFC 2827, May 2000.

[13] Mark Fullmer and Steve Romig. The OSU Flow-tools Package and Cisco Netflow logs. In *Proceedings of the 2000 USENIX LISA Conference*, New Orleans, LA, December 2000.

[14] Virgil Gilgor. A Note on the Denial-of-Service Problem. In *Proceedings of the 1983 IEEE Symposium on Security and Privacy*, Oakland, CA, 1983.

[15] John D. Howard. *An Analysis of Security Incidents on the Internet*. PhD thesis, Carnegie Mellon University, August 1998.

[16] Phil Karn and William Simpson. Photuris: Session-Key Management Protocol. RFC 2522, March 1999.

[17] David Moore, Ram Periakaruppan, Jim Donohoe, and kc claffy. Where in the World is netgeo.caida.org? In *INET 2000 Proceedings*, June 2000.

[18] Roger Needham. Denial of Service: An Example. *Communications of the ACM*, 37(11):42–47, November 1994.

[19] V. Paxson, G. Almes, J. Mahdavi, and M. Mathis. RFC 2330: Framework for IP performance metrics, May 1998.

[20] Vern Paxson. Personal Communication, January 2001.

[21] Jon Postel, Editor. Internet Control Message Protocol. RFC 792, September 1981.

[22] Steve Romig and Suresh Ramachandran. Cisco Flow Logs and Intrusion Detection at the Ohio State university. *login; magazine*, pages 23–26, September 1999.

[23] Stefan Savage, David Wetherall, Anna Karlin, and Tom Anderson. Practical Network Support for IP Traceback. In *Proceedings of the 2000 ACM SIGCOMM Conference*, pages 295–306, Stockholm, Sweden, August 2000.

[24] Dawn Song and Adrian Perrig. Advanced and Authenticated Marking Schemes for IP Traceback. In *Proceedings of the 2001 IEEE INFOCOM Conference*, Anchorage, AK, April 2001.

[25] Oliver Spatscheck and Larry Peterson. Defending Against Denial of Service Attacks in Scout. In *Proceedings of the 1999 USENIX/ACM Symposium on Operating System Design and Implementation*, pages 59–72, February 1999.

[26] Robert Stone. CenterTrack: An IP Overlay Network for Tracking DoS Floods. In *Proceedings of the 2000 USENIX Security Symposium*, pages 199–212, Denver, CO, July 2000.

MULTOPS: a data-structure for bandwidth attack detection

Thomer M. Gil and Massimiliano Poletto
Vrije Universiteit, Amsterdam, The Netherlands
and
M.I.T., Cambridge, MA, USA
{thomer,maxp}@lcs.mit.edu

Abstract

A denial-of-service bandwidth attack is an attempt to disrupt an online service by generating a traffic overload that clogs links or causes routers near the victim to crash. We propose a heuristic and a data-structure that network devices (such as routers) can use to detect (and eliminate) such attacks. With our method, each network device maintains a data-structure, *MULTOPS*, that monitors certain traffic characteristics. MULTOPS (MUlti-Level Tree for Online Packet Statistics) is a tree of nodes that contains packet rate statistics for subnet prefixes at different aggregation levels. The tree expands and contracts within a fixed memory budget.

A network device using MULTOPS detects ongoing bandwidth attacks by the significant, disproportional difference between packet rates going to and coming from the victim or the attacker. MULTOPS-equipped routing software running on an off-the-shelf 700 Mhz Pentium III PC can process up to 340,000 packets per second.

1 Introduction

A bandwidth attack is an attempt to disrupt an online service by generating a traffic overload that clogs links or causes routers near the victim to crash. This can have serious consequences for Web companies which rely on their online availability to do business. This paper introduces a data-structure that routers and network monitors can use to collect packet rate statistics for subnet prefixes at different aggregation levels. These statistics can be used to detect bandwidth attacks using a simple heuristic: a significant, disproportional difference between the packet rate going to and coming from a host or subnet. This heuristic is based on the assumption that, during normal operations on the Internet, the packet rate of traffic going in one direction is proportional to the packet rate of traffic going in the opposite direction. Although this assumption does not hold in some cases, it is a close approximation to reality.

Bandwidth attacks are typically distributed attacks. An attacker uses tools to gain root access to machines on the Internet [Pac00, Spi00]. Once a machine is cracked, it is turned into a "zombie." The attacker instructs the zombies to send bogus data to one particular destination [Dit00]. The resulting traffic can clog links, cause routers near the victim or the victim itself to fail under the load.

One major reason underlies the absence of a simple solution against bandwidth attacks: attackers can release high volumes of normal-looking packets on the Internet without being conspicuous or easily traceable. It is the mass of all packets together directed at one victim that poses a threat, rather than any characteristics of the individual packets. A dropping policy in routers based on per-packet characteristics will, therefore, not work.

It is relatively easy, but rather useless, to detect a bandwidth attack in the vicinity of the victim: by measuring the traffic load on a link or in a router, the exceptionally high volume of packets can be detected. Unfortunately for the victim, determining that it is under attack will not make the packets go away. Harm has already been done by the time the malicious packets reach (the vicinity of) the victim. A bandwidth attack should, therefore, be detected close to the attacker rather than close to the victim so that malicious packets can be stopped before they can cause any harm.

This paper proposes a *MUlti-Level Tree for Online Packet Statistics (MULTOPS)*. MULTOPS enables routers or network monitors to detect ongoing bandwidth attacks. A handful of attackers that blast packets to a victim without any (or disproportionally fewer) packets coming back will be identified as malicious by MULTOPS. Large attacks that occurred in February 2000 [CNN00a, CNN00b, Net00] displayed these disproportional packet flows. Routers (or network monitors) using MULTOPS could have been used to stop (or detect) those attacks.

MULTOPS is a tree of nodes that contains packet rate statistics for subnet prefixes at different aggregation levels. It dynamically adapts its shape to (1) reflect changes in packet rates, and (2) avoid (maliciously intended) memory exhaustion.

Depending on their setup and depending on their location on the network, MULTOPS-equipped routers or network monitors may fail to detect a bandwidth attack that is mounted by attackers that randomize IP source addresses on malicious packets. In a different setup, MULTOPS-equipped routers may cause "collateral damage" by dropping legitimate packets with an IP destination address that MULTOPS identified as being under attack.

MULTOPS fails to detect attacks that deploy a large number of proportional flows to cripple a victim. (Proportional flows are flows in which the packet rate in one direction is proportional to the packet rate in the opposite direction.) For example, many attackers could open FTP or HTTP connections to one victim and download—preferably large—files over these connections, thereby overloading the victim. Even though the packet rates between the attackers and the victim are relatively low (because the victim cannot handle all the parallel downloads), they are proportional and, therefore, undetectable by MULTOPS. However, to successfully mount such an undetectable bandwidth attack, attackers need to be numerous, geographically distributed, and well organized. This makes it more difficult to mount an undetectable attack.

MULTOPS has been implemented in a software router and was tested with simulated attacks. Results are encouraging: attacks are stopped and legitimate traffic continues in a normal fashion, even with a large number of participating attackers. An off-the-shelf 700 Mhz Pentium III PC, running MULTOPS-equipped routing software, routes between 240,000 to 340,000 packets per second, depending primarily on the resources available to MULTOPS.

The rest of this paper is organized as follows. Section 2 takes a look at related work, Section 3 looks at different types of bandwidth attacks, Section 4 explains the design of MULTOPS, Section 5 looks at the details of the MULTOPS implementation, Section 6 deals with measurements, Section 7 discusses the details of some (unresolved) issues, and Section 8 concludes this paper.

2 Related work

Most of the techniques proposed so far for protection against denial-of-service attacks can be used in conjunction with MULTOPS. We quickly review the major techniques and how MULTOPS can augment them.

Ingress/egress filtering is a technique performed by routers to effectively eliminate IP spoofing [ea00, Ins00]—lying about one's own IP address in the header of outgoing IP packets. To stop spoofed IP packets, edge routers match the IP source address of each outgoing packet against a fixed set of known IP address prefixes. If no match is found, the packet is dropped. Another possible technique is for a router to only send off a packet from interface i if a potential reply to this packet is, according to the router's routing tables, expected to arrive on interface i. If not, the packet is dropped. Even though these techniques are simple and effective remedies against IP spoofing, unfortunately many routers are not configured to deploy these techniques and they are not complete solutions. However, MULTOPS benefits from them because IP spoofing hurts MULTOPS' ability to detect attacks (see also Section 7.1).

IP Traceback assists in tracking down attackers post-mortem [SWKA00, SP01, DFS01]. This technique requires routers to, with a low probability, mark packets such that the receiving end can reconstruct the route that packets followed, provided enough packets were sent. A similar technique is ICMP Traceback [Bel00]. When forwarding packets, routers can, with a low probability (1/20,000), generate an ICMP Traceback message that is sent along to the destination. With enough traceback messages from enough routers along the path, the traffic source and path can be determined. The main advantage of these techniques is that it assists in finding attackers. It does not stop them.

All the traceback approaches have serious deployment and operational challenges. A sufficient number of routers need to support traceback before it is effective. Attackers can generate traceback messages too, so some form of authentication of traceback messages is necessary. The victim of a bandwidth attack might also not receive enough traceback messages because they might get dropped by overloaded routers. In addition, if an attack is very distributed, there may not be enough traceback information to find the attackers.

A number of routers provide information about packets that can be used to implement the same detection heuristic that MULTOPS is using. Cisco routers, for example, support RMON [Cisb] and Netflow [Cisa]. Unfortunately, both RMON and Netflow data is expensive to process off-line. RMON copies complete packets to a port for off-line analysis—this slows down the router's normal operation. Netflow keeps a table with 45-byte entries for every flow, which can be queried by and transferred to an external analysis program. Netflow provides no protection against attackers that might blow up the table. In the worst case, RMON and Netflow can magnify an attack. MULTOPS is intended to be integrated into a router or a monitoring device for on-line analysis. MULTOPS also runs in a fixed-size memory footprint so that attackers cannot run a MULTOPS device out of memory.

Stone [Sto99] proposes CenterTrack, an overlay network that consists of IP tunnels which can be used to selectively reroute packets from routers on a network to special "tracking" routers. This architecture can be used to analyze traffic for signs of a bandwidth attack, and optionally drop traffic that seems suspicious. MULTOPS could probably be used as a component of CenterTrack to help routers determine whether a bandwidth attack is occurring and what IP addresses are involved.

Bellovin [Bel01] discusses aggregate congestion control and pushback. The central idea is to identify "aggregates"—subsets of traffic defined by some characteristic, such as a particular destination address—that may be involved in the bandwidth attack. Pushback is a cooperative mechanism in which routers can ask adjacent routers to block an aggregate upstream. MULTOPS could be viewed as a data-structure for efficiently tracking the aggregate defined by IP addresses for which traffic flow is asymmetric.

Intrusion detection system such as Bro [Pax99] try to detect attacks by monitoring network links over which the attacker's traffic transits. Armed with (statistical) knowledge about normal behavior of different applications and protocols, these systems detect anomalies in traffic patterns and report a wide range of attack types. Although similar to MULTOPS in that it monitors traffic, the primary difference is that these systems do not attempt to stop attacks.

3 Bandwidth attacks

The common denominator of all bandwidth attacks is the desire to cripple someone else's infrastructure by generating a traffic overload. Bandwidth attacks vary, among other things, in the protocol being used to mount the attack. In addition, attackers can use IP spoofing. As mentioned above, IP spoofing is lying about one's own IP address.

Since routing is done based on the IP destination address only, the IP source address can be anything. In some cases, attackers use one specific forged IP source address on all outgoing IP packets to make all returning IP packets—and possibly ICMP messages—go to the unfortunate owner of that address. Attackers also use IP spoofing to hide their location on the network. Section 7.1 discusses how IP spoofing affects MULTOPS' ability to detect (the source(s) of) attacks.

An attacker can forge an ICMP packet with a spoofed IP source address and launch a "Smurf" attack [CC98]: he sends this *one* forged ICMP packet to a broadcast address and *all* the receivers respond with a reply to the spoofed IP address (the victim). (A solution is to never reply to ICMP packets that are sent on a broadcast address, or to let routers filter such packets [ea81, ea00].) In a "Fraggle" attack, an attacker instructs many zombies to send UDP packets to one victim. Both Smurf and Fraggle attacks can be detected by MULTOPS because in both cases the packet rate to the victim exceeds the packet rate coming back from the victim in a disproportional manner.

There are several types of attack that use TCP. The best known is "SYN Flooding" [CC96]. Several solutions have been proposed for solving SYN Floods: lowering the TCP time-out, increasing the number of TCP control blocks, SYN cookies [AH99] that eliminate the need to store information on half-open connections, and special firewalls that buffer SYN packets. Although a SYN Flood is actually a resource attack, it is similar to a bandwidth attack because of the flood of SYN packets.

Another attack works by generating a huge amount of normal traffic by, for example, running a JavaScript program in a browser that pops up a few dozen windows each fetching a Web page from one server. This may constitute a problem if a few thousand people are willing to run this script in their browser simultaneously [ec00]. Such a script could easily spread by means of self-replicating e-mail viruses. (This phenomenon can also occur without it being an attack.)

As mentioned in Section 1, attacks that cripple a victim by sending or receiving a high volume of traffic using proportional flows may go unnoticed by MULTOPS.

4 MULTOPS design

4.1 Overview

MULTOPS uses disproportional packet rates to or from hosts and subnets as a heuristic to detect (and potentially stop) attacks. To collect these statistics, a tree-shaped data-structure keeps track of packet rates to and from those subsets of the IP address space that display disproportional behavior. This is done by letting the tree expand and contract ("zoom in and zoom out") based on observed (disproportional) traffic patterns.

MULTOPS stores packet rate statistics for flows between hosts (or subnets) A and B using either A's IP address or B's IP address. As a consequence, MULTOPS can either establish the victim, or the source(s) of the attack. We distinguish between these two modes by defining them as *victim-oriented* mode and *attacker-oriented* mode, respectively. In victim-oriented mode, MULTOPS tries to identify the IP address of the victim of an attack. In attacker-oriented mode, MULTOPS tries to identify the IP address(es) of the attacker(s). The difference between these two modes becomes important when dropping packets: either packets going to the victim are dropped, or packets coming from the attacker are dropped. Note that in both cases the attack is stopped. In one case this is done based on the IP address of the victim, in the other case it is done based on the IP address(es) of the attacker(s). Throughout this paper we assume that MULTOPS runs in victim-oriented mode, unless specified otherwise.

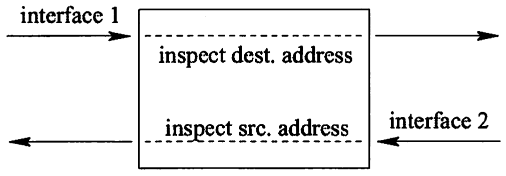


Figure 1: Schematic MULTOPS in victim-oriented mode

MULTOPS expects two streams of IP packets as input—each connected to a different network interface. Packets going in one direction ("forward packets") are inspected on their destination address; packets going in the opposite direction ("reverse packets") are inspected on their source address. Figure 1 illustrates this. Exchanging the network interfaces switches between attacker-oriented and victim-oriented mode.

MULTOPS presents a query interface that returns an approximation to $R(P)$. $R(P)$ is the ratio of forward packets with destination IP address prefix P to reverse packets with source IP address prefix P.

In victim-oriented mode, MULTOPS determines a victim's IP address by looking for prefixes for which $R(P)$ is greater than some threshold. Dropping packets with destination addresses matching such prefixes might defeat the attack, though it may also impose "collateral damage" by dropping legitimate packets. In attacker-oriented mode, MULTOPS determines the addresses of attackers by looking for prefixes for which $R(P)$ is less than some threshold. Dropping packets based on source addresses matching such prefixes might defeat the attack, though IP spoofing introduces complications that are discussed in Section 7.1. Note that a single MULTOPS cannot detect both attacker and victim addresses.

In our current design, we also assume that packets are being sent using IPv4. Our approach should easily extend to IPv6, although it will consume significantly more resources.

4.2 MULTOPS heuristic

Packets are defined to be malicious (and, thus, may be dropped) if they are destined for a host or subnet from which too few packets are coming back. This heuristic is based on the assumptions that (1) most Internet traffic consists of packet flows, and (2) during normal operations, the rate of packets in a flow going from A to B is proportional to the packet rate going from B to A. Thus, during normal operations on the Internet, the packet rate of traffic going in one direction is proportional to the packet rate of traffic going in the opposite direction. If not, something must be wrong.

This heuristic appears to hold broadly. TCP, the protocol mainly used on the Internet, acknowledges every single—or every k—received packets by sending back a packet, and, therefore, has proportional packet flows.

The following example illustrates the heuristic. If machine A is sending legitimate TCP packets to machine B, but B is suffering under a bandwidth attack, then A's packets will not reach B. Even *if* some of A's packets reach B, then B's packets may not reach A because of the overloaded links and routers. In reaction to the absence of B's packets, A will automatically decrease the sending rate and, eventually, stop sending packets to B altogether. If, on the other hand, A is an attacker that blasts (any type of) packets at B, a MULTOPS-equipped router routing A's packets to B will detect the disproportional packet rates between them and could decide to drop packets going to B. Consequently, B will not have to cope with A's packets.

Let $R(P)$ be the ratio between the packet rate going to and coming from addresses with prefix P. Under normal circumstances, R is close to some constant k for all P, i.e., packet rates are proportional for all prefixes. If R drops below R_{min} or exceeds R_{max}, then a (host in) subnet with prefix P is either under attack or a subnet with prefix P harbors an attacker.

MULTOPS collects packet rates to and from address prefixes so that, given a certain P, $R(P)$ can be calculated. Packets may be dropped if they are destined for a host or subnet from which disproportionally fewer packets are coming back, i.e., if $R(P)$ is not between R_{min} and R_{max}. The sensitivity of MULTOPS can be tuned by changing the values of R_{min} and R_{max}.

4.3 Data structure

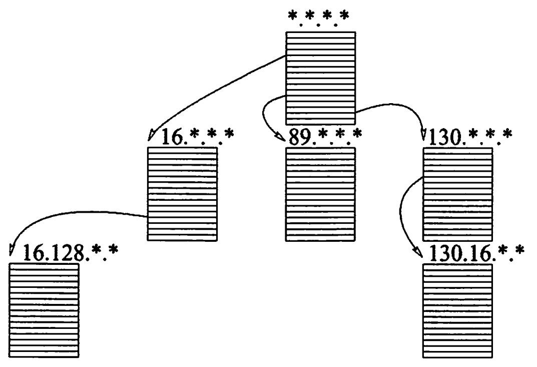


Figure 2: MULTOPS

MULTOPS is organized as a 4-level 256-ary tree to conveniently cover the entire IPv4 address space. Each node in the tree is a table consisting of 256 records, each of which consists of 3 fields: 2 rates—to-rate and from-rate—and 1 pointer potentially pointing to a node in the next level of the tree. A table stores all packet rates to and from IP addresses with a common 0-bit, 8-bit, 16-bit, or 24-bit prefix, depending on the level of the tree. Deeper levels of the tree contain packet rates for addresses with a longer prefix. Thus, the root node contains the aggregate packet rates to and from address 0.*.*.*, 1.*.*.*, 2.*.*.*, etc. The 90th record in the root node, for example, contains the packet rates to and from addresses with 8-bit prefix 89, and a pointer to a node that keeps tracks of the aggregate packet rates to and from addresses with that prefix, i.e., 89.0.*.*, 89.1.*.*, 89.2.*.*., etc. The sum of all 256 to-rates and the sum of all 256 from-rates in a node are equal to the to-rate and the from-rate in the parent record of that node. Figure 2 shows a sample MULTOPS.

When the packet rate to or from a subnet reaches a certain threshold, a new subnode is created on the fly to keep track of more fine-grained packet rates, potentially down to per-IP address packet rates. For example, if the aggregate packet rate to or from subnet 130.17.*.* exceeds R_{max}, a new node is created to keep track of packet rates to and from subnets 130.17.0.*, 130.17.1.*, etc. Creating new nodes is called *expansion*. The reverse, i.e., removing nodes or entire subtrees, is called *contraction*. Contraction is done when the packet rate from and to a given IP address prefix drop below a certain threshold, or when memory is running out, possibly due to a memory exhaustion attack against MULTOPS itself.

Expansion and contraction enable MULTOPS to exploit the hierarchical structure of the IP address space and the fact that a bandwidth attack is usually directed at (or coming from) a limited set of IP addresses—with a common prefix—only. MULTOPS detects the attack on a high level in the tree (where prefixes are short) and expands toward the largest possible common prefix of the victim's IP address(es), potentially establishing single IP address(es) that are under attack.

4.4 Algorithm

Each packet (or every nth packet) that is routed causes packet rates in applicable nodes in the tree to be updated; starting in the root, and going down to the deepest available node. This works as follows. The first byte of the IP *destination* address of a *forward packet* is used as an index in the root node to find the record in which to update the *to-rate*. For *reverse packets* the first byte of the IP *source* address is used as an index in the root node to find the record in which to update the *from-rate*. If the record has a child, the process descends down to the child and continues. If no child exists, it is created if either the from-rate or the to-rate exceeds a certain threshold. In any case, the process may follow the pointer in the record to the child node. In this child node, the *second* byte of the IP address is used as an index to find the record and update the packet rates. This process may descend down to the deepest level in the tree where per-IP address packet rates are kept. The full algorithm is given in pseudo-code in Algorithm 4.1.

Algorithm 4.1: UPDATE(*addr*, *packet*, *fwd*)

```
TABLE t ← root
for i ← 1 to 4
  do { RECORD r ← t[addr[i]]
       if fwd
         then update r's to-rate
         else update r's from-rate
       if r has no child node
         then break
       t ← r's child node
annotate packet with r's from-rate and to-rate        (1)
if (r's from-rate > threshold
 or r's to-rate > threshold)
 and t is not a node in deepest level of tree
  then create child table t' under r
```

Method UPDATE() is called by method HANDLE_PACKET() described in Section 5.2. Parameter *addr* is the 4-byte IP source or destination address of packet *packet*, depending on whether MULTOPS is set up in victim-oriented or attacker-oriented mode. Parameter fwd tells UPDATE() whether this packet is a forward packet or a reverse packet. Statement 1 immediately after the **for** -loop annotates the packet with r's from-rate and to-rate. This annotation can later be used by a part of the system that implements the heuristic to determine whether or not this packet is part of a malicious flow and should, thus, be dropped.

4.5 Expansion and contraction

If the to-rate or the from-rate for an address with an n-bit prefix P exceeds the *expand threshold*, MULTOPS creates a child node under the record for prefix P to keep track of packet rates for addresses with $(n+8)$-bit prefix P'. Lowering this expand threshold increases precision of MULTOPS, but also increases its memory use. Figure 3 shows how a new node is added to the tree to keep track of all packet rates to and from addresses with prefix 130.16.120.

The reverse of expansion is contraction. Contracting a record involves removing a subtree from under a record. A subtree is contracted when the aggregate packet rate for that subtree drops below R_{max}. Contraction is done to constrain memory use and to avoid (maliciously intended) memory exhaustion. Figure 3 shows how a node is contracted.

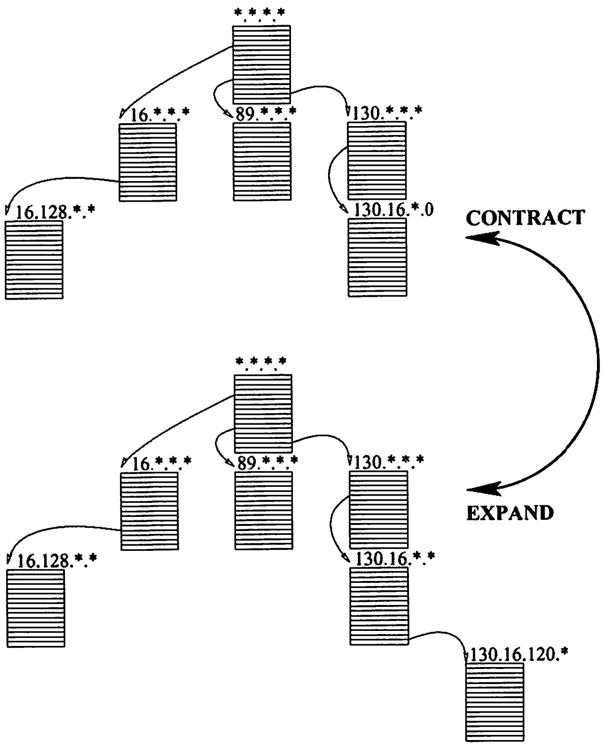


Figure 3: Expansion and contraction

Traversing the entire tree in search of subtrees to contract is potentially expensive and its frequency should be chosen with care. Traversing the tree for every x routed packets is dangerous because a router should have its resources free for routing, not for contracting when packet rates go up. Traversing the tree every t ms is safer, but choosing t correctly is tricky: if t is too high, memory might run out before traversal starts. The strategy we chose is to never allocate more memory than a certain limit m—thereby making memory exhaustion impossible—and to traverse the tree every t ms in search of subtrees to contract. In the time period between reaching memory limit m and the next "cleanup," MULTOPS cannot create new nodes. It is, therefore, important to choose t low, but not so low as to trigger cleanups too often and, thus, waste the router's resources.

An attacker might try to launch a memory exhaustion attack against MULTOPS by causing it to branch profusely. The two opposing forces are the attacker causing nodes to be created versus contraction causing nodes to be destroyed. Since a subtree is contracted when the packet rates to and from addresses with a certain prefix are less than the expand threshold, the attacker will have to sustain a higher packet rate for as many different address prefixes as possible. Section 5.4 deals with this issue in a quantitative context.

5 MULTOPS implementation

MULTOPS is implemented using Click [KMC+00]. Click is a modular software router architecture developed at the MIT Laboratory for Computer Science. A Click router is an interconnected collection of modules called elements. Each element performs a simple, straightforward task such as communicating with devices, queueing packets, and implementing a dropping policy. Each element has 0 or more inputs and 0 or more outputs. Inputs are used to receive packets from other elements. Outputs are used to hand off packets to other elements. Configuration of a Click router is done by feeding it a file describing which elements to use and how the inputs and outputs of these elements interconnect.

MULTOPS is implemented as 2 separate elements: `IPRateMonitor` and `RatioBlocker`. Adding these elements to the configuration adds the MULTOPS detection mechanism and the related dropping policy to the router. `IPRateMonitor` tags each packet with from-rate and to-rate such that `RatioBlocker` may decide to drop the packet based on these tags and based on the defined thresholds (i.e., R_{min} and R_{max}). Thus, `IPRateMonitor` implements the tree, `RatioBlocker` implements a dropping policy based on the MULTOPS detection heuristic.

`IPRateMonitor` has 2 inputs and 2 outputs. Each input should be connected to a different physical network interface. `RatioBlocker` has 1 input and 1 output.

5.1 Data structure

`IPRateMonitor` is a C++ class that defines two private `structs`: `Record` and `Table`. Figure 5.1 contains the C++ code that defined these `structs`.

`from_rate` and `to_rate` in `Record` are used to store packet rates. `EWMA` implements an exponentially weighted moving average and is used to keep track of rates. `child` contains a pointer to a child or `NULL` if no child exists. Besides 256 pointers to `Record`, `Table` contains a pointer to the parent record (`parent`) and two pointers (`prev` and `next`) that are used to maintain a doubly-linked list of nodes—their use is explained in Section 5.3. `root` points to the root node.

```
struct Record {
  EWMA from_rate;
  EWMA to_rate;
  Table *child;
};

struct Table {
  Record *parent;
  Table *prev, *next;
  Record* record[256];
};

Table *root;          // root node
```

Figure 4: C++ code that defines `Record` and `Table`

5.2 Algorithm

`IPRateMonitor`'s method HANDLE_PACKET() (given in pseudo-code in Algorithm 5.1) implements the functionality represented by Figure 1. It is, together with method UPDATE(), responsible for implementing the algorithm described in Section 4.4.

Algorithm 5.1: HANDLE_PACKET(*port*, *packet*)

```
if port == 0
  then UPDATE(packet.dest_addr, packet, true)
  else UPDATE(packet.src_addr, packet, false)
```

`IPRateMonitor`'s 2 input ports should each be logically connected to one of the network interfaces. Port 0 connects to the interface for forward packets, port 1 connects to the interface for reverse packets. (This is achieved through Click configuration.) *port* is the input of the `IPRateMonitor` element that packet *packet* arrived on. This information is passed to UPDATE() using its fwd parameter.

5.3 Expansion and contraction

In addition to the tree itself, MULTOPS maintains a doubly-linked list of pointers to nodes in the tree using `prev` and `next` in `Table`. Each time a new node is created in the tree, i.e., expansion occurs, a pointer to that node is added at the end of the linked list. During a cleanup, the list is traversed. A node (and all its children) is deleted when the sum of all from-rates and the sum of all to-rates in that node are both lower than the expand threshold. (Both sums are, by definition, stored as from-rate and to-rate in the parent record of that node; hence the need for the `parent` pointer in `Table`.) The root node is never deleted. The list is either traversed backwards or forwards to avoid checking the same nodes every time thereby causing starvation-like phenomena.

To avoid heavy memory fluctuations and to avoid spending too much time on a single cleanup, contraction stops when a certain fraction f of all allocated memory has been freed. If none of the nodes can be deleted, but memory is at its imposed maximum, then some nodes *must* be deleted. In that case, the expand threshold is decreased by some factor and the cleanup starts again. This may have to be repeated multiple times until fraction f of all memory has been freed.

5.4 Memory exhaustion attacks

To defeat our mechanism, an attacker may try to exhaust a router's memory by making `IPRateMonitor` allocate many nodes. (Of course, memory exhaustion is only possible when `IPRateMonitor` has no imposed memory limit.) An attacker achieves this by sending packets with a wide variety of spoofed IP source addresses through that router. (This is a problem only when MULTOPS is in attacker-oriented mode.) Each stream of packets with a common IP source address needs to have a bandwidth higher than the expand threshold of MULTOPS—otherwise MULTOPS contracts the nodes, thereby defeating the attacker's goal to run it out of memory. If an attacker is not bound by any resource constraints, nor by ingress/egress filtering, he can create a worst-case scenario by sending spoofed IP packets such that the number of nodes in MULTOPS is maximized.

Given the structure of the MULTOPS tree, the size of a `Table` (1040 bytes), the size of a `Record` (28 bytes), a packet size of 34 bytes, and an expand threshold of 1000 packets per second, an attacker, launching such a worst-case scenario memory exhaustion attack, needs to generate traffic with a bandwidth of roughly 16 Gbit/s to make `IPRateMonitor` allocate 128MB of memory, provided that the network has the physical capability to carry this traffic to the target router. This number was derived by calculating the amount of allocated memory based on the number of different address prefixes stored in the tree. The expand threshold can be set to a value that ensures that memory will never run out. It is safe to conclude that, even without an imposed memory limit, it is impossible to run `IPRateMonitor` out of memory.

6 Measurements

To measure the performance of `IPRateMonitor`, a simple Click configuration was run in a Linux kernel 2.2.16 on an off-the-shelf PC (700 Mhz Pentium III, 256 KB cache, 256 MB memory) that sends packets through an `IPRateMonitor` element. Bogus UDP packets were generated by Click itself to avoid time consuming interaction with network interfaces. IP spoofing attackers were simulated by generating UDP packets with an IP source address picked from a fixed set of IP addresses in round-robin fashion. Measurements were done for different memory limits and for an expand threshold of 0, i.e., maximum expansion.

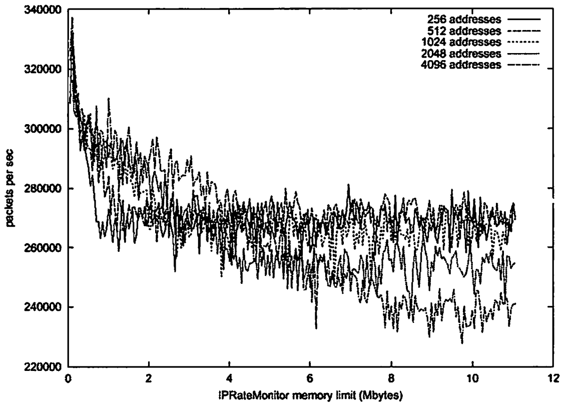


Figure 5: Packet rate as a function of memory limit

The graph in Figure 5 shows the number of packets that `IPRateMonitor` can handle as a function of its imposed memory limit. The graph shows this for 5 UDP flood attacks that differ only in the number of attackers (i.e., IP source addresses) involved. The IP source addresses used in the malicious UDP packets constitute a worst-case scenario (see Section 5.4).

The graph shows that `IPRateMonitor` performs better when it has little memory at its disposal. A small tree fits in cache entirely and is, therefore, fast. When more memory is available, the tree size increases up to the point where it is too big to fit in cache, and cache misses result. The performance of `IPRateMonitor` for 256, 512, and 1024 addresses is roughly the same (270,000 packets/sec), because in these cases the tree is small enough to fit in cache entirely. For 2048 and 4096 addresses, rates drop proportional to the total memory consumption of the tree, up to the point where the tree reaches its maximum size, after which memory consumption—and thus performance—fluctuates around the same point.

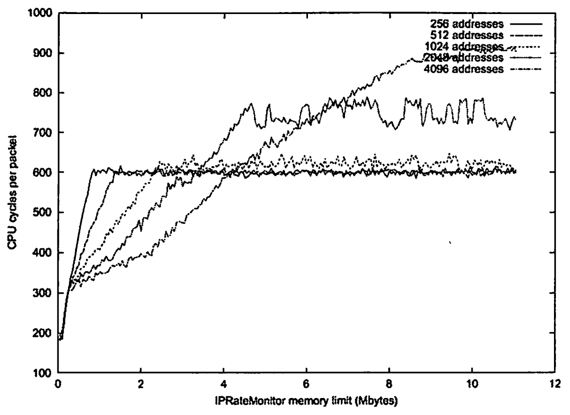


Figure 6: CPU cycles per packet as a function of memory limit

The graph in Figure 6 shows the number of CPU cycles that `IPRateMonitor` consumes per packet as a function of its imposed memory limit. `IPRateMonitor` consumes more CPU cycles when it has more memory at its disposal. These extra cycles are, most likely, spent on waiting for a memory fetch after a cache miss. Unsurprisingly, the graph in Figure 6 is essentially the reciprocal of the graph in Figure 5.

`IPRateMonitor` performs better when it has little memory at its disposal. Unfortunately, its ability to expand and, therefore, to precisely determine the source(s) and/or target(s) of the attack, is also more limited. Thus, the tradeoff is precision vs. performance.

7 Discussion

7.1 IP spoofing

MULTOPS in victim-oriented mode is not influenced by IP spoofing. However, MULTOPS may impose "collateral damage" by dropping legitimate packets going to the victim.

When attackers randomize IP source addresses—a common practice—then a problem arises for MULTOPS in attacker-oriented mode. There could be so many different (spoofed) IP source addresses that MULTOPS does not have enough available memory to establish all "malicious" IP source addresses. In that case, MULTOPS *can* establish a set of prefixes that malicious IP source addresses share. Better randomization implies shorter address prefixes. Shorter prefixes implies that MULTOPS drops more packets, which may include legitimate packets. In other words: collateral damage as a result of MULTOPS' dropping policy is greater when IP spoofing gets more randomized.

When attackers *perfectly* randomize IP source addresses, each malicious stream of packets with a common IP source address (or prefix) is either too insignificant to be seen as part of an attack, or *all* malicious streams are seen as part of an attack. In the former case, MULTOPS does not detect the attack at all. In the latter case, all packets are considered part of an attack, and, hence, dropped. Both cases constitute a successful denial-of-service attack.

7.2 Distribution

The IP spoofing problem described above closely relates to the problem of attacker distribution. As more (spoofing or non-spoofing) attackers participate in a bandwidth attack, it becomes harder (for MULTOPS in attacker-oriented mode) to identify a single attacker because its relative share in the total mass becomes smaller and, therefore, the disproportional quality of the traffic less conspicuous.

When a total number of T packets per second is required to crash the victim's infrastructure, and N attackers participate, then each attacker needs to generate an average of T/N packets per second. As N gets larger, T/N gets smaller.

Even though MULTOPS' sensitivity can be tuned, if N is too large and, consequently, T/N too small, one single attacker might go undetected by MULTOPS. If, though, attackers do not spread out geographically, their combined generated traffic might go through a single MULTOPS-equipped router that could decide to drop all the packets. Even *if* the attackers are perfectly distributed throughout the world, the malicious packets get funneled on their way to the victim by routers. The chance of being detected as a malicious stream by one of these routers gets larger as the stream gets more bundled (and, thus, packet rates become more disproportional).

7.3 Different protocols

MULTOPS relies on the assumption that, during normal operations, packet rates between two communicating parties are proportional. There are, however, different protocols, each with different implementations. With

TCP, for example, implementations differ in their acknowledgment policy, although most TCP implementations acknowledge at least every other packet. Nonetheless, defining the MULTOPS detection heuristic quantitatively, i.e., choosing suitable values for R_{min} and R_{max}, is tricky. In the current implementation of `RatioBlocker`, $R_{min} = 0.66$, and $R_{max} = 2.5$. These values were experimentally determined. One can imagine implementing a `RatioBlocker` that adjusts these values based on observed traffic patterns during normal operations, making the heuristic more flexible.

Protocols such as UDP and ICMP do not require acknowledgments at all. However, several applications such as NFS and DNS display proportional behavior similar to TCP, which is advantageous for the MULTOPS detection heuristic. Since most services on the Internet are TCP-based, we suggest rate-limiting all non-TCP traffic during an attack. Even though this is a drastic measure, it will allow most Internet traffic to proceed normally.

7.4 Asymmetric routes

MULTOPS needs to see traffic in both directions to detect disproportional packet rates—this requires symmetric routes. However, Paxson demonstrated that many routes on the Internet are asymmetric [Pax97]. To circumvent this problem, MULTOPS should be placed on the edges of the network—in a data center, for example. If such a site is multi-homed, then packet rate statistics from all on-site routers need to be combined. This requires (preferably out of band) communication between several MULTOPS-equipped routers. The details of such a setup are beyond the scope of this paper.

7.5 Granularity

When MULTOPS has more memory at its disposal, it can expand to deeper levels, thereby increasing its precision. Dropping packets based on disproportional packet rates in a record in the root node will affect many machines, i.e., all machines with a common first byte in their IP address. If, however, dropping packets is done based on disproportional packet rates from/to a single IP address—stored in the deepest level of the tree—then only the machine with that IP address will be affected. It is, therefore, important to not restrict MULTOPS' memory use too much.

8 Conclusion

This paper proposes MULTOPS. MULTOPS enables routers or network monitors to detect ongoing bandwidth attacks using a simple heuristic: a significant, disproportional difference between the packet rate going to and coming from a host or subnet. This is based on the assumption that, during normal operations on the Internet, the packet rate of traffic going in one direction is proportional to the packet rate of traffic going in the opposite direction.

MULTOPS is a tree of nodes that contains packet rate statistics for subnet prefixes at different aggregation levels. It dynamically adapts its shape to (1) reflect changes in packet rates, and (2) avoid (maliciously intended) memory exhaustion.

MULTOPS successfully detects bandwidth attacks that create disproportional packet flows between the sender(s) and the receiver. To our knowledge, no such detection mechanism has been proposed yet. Depending on the situation, MULTOPS can point out the source(s) of the attack.

MULTOPS is not a complete solution against bandwidth attacks. However, it enables network devices to maintain statistics to establish whether or not a bandwidth attack may be going on.

Measurements show that the performance of MULTOPS is primarily influenced by the size of the cache and the number of IP source addresses involved in the attack. It is exceedingly difficult to run a MULTOPS-equipped router out of memory.

9 Acknowledgements

We would like to thank Frans Kaashoek, Robert Morris, Andrew Tanenbaum, and the entire PDOS team for their help.

References

[AH99] Thamer Al-Herbish. Secure Unix Programming FAQ, 1999. Available at http://www.whitefang.com/sup/secure-faq.html.

[Bel00] Bellovin. ICMP Traceback Messages. Technical report, AT&T, 2000. Available at http://www.ietf.org/internet-drafts/draft-bellovin-itrace-00.txt.

[Bel01] Steven Bellovin. DDoS Attacks and Pushback, February 2001. NANOG 21, Atlanta, GA, USA.

[CC96] CERT Coordination Center. CERT Advisory CA-1996-21 TCP SYN Flooding and IP Spoofing Attacks, 1996. Available at http://www.cert.org/advisories/CA-1996-21.html.

[CC98] CERT Coordination Center. CERT Advisory CA-98.01 "smurf" IP Denial-of-Service Attacks, 1998. Available at http://www.cert.org/advisories/CA-98.01.smurf.html.

[Cisa] Cisco. Netflow Services and Applications. Available at http://www.cisco.com/warp/public/732/netflow/.

[Cisb] Cisco. RMON. Available at http://www.cisco.com/warp/public/614/4.html

[CNN00a] CNN. Cyber-attacks batter Web heavyweights, February 2000. Available at http://www.cnn.com/2000/TECH/computing/02/09/cyber.attacks.01/index.htm%1.

[CNN00b] CNN. 'Immense' network assault takes down Yahoo, February 2000. Available at http://www.cnn.com/2000/TECH/computing/02/08/yahoo.assault.idg/index.ht%ml.

[DFS01] Drew Dean, Matt Franklin, and Adam Stubblefield. An Algebraic Approach to IP Traceback. In *Proceedings of the 2001 Network and Distributed Systems Security Symposium*, February 2001.

[Dit00] Dave Dittrich. The DoS Project's 'trinoo' distributed denial of service attack tool. Technical report, University of Washington, 2000. Available at http://staff.washington.edu/dittrich/misc/trinoo.analysis.txt.

[ea81] J. Postel et al. RFC 792. Internet Control Message Protocol. Technical report, IETF, 1981. Available at http://sunsite.cnlab-switch.ch/ftp/doc/standard/rfc/7xx/792.

[ea00] P. Ferguson et al. RFC 2827. Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing. Technical report, IETF, February 2000. Available at http://sunsite.cnlab-switch.ch/ftp/doc/standard/rfc/28xx/2827.

[ec00] The electrohippies collective. Client-side Distributed Denial-of-Service, 2000. Available at http://www.gn.apc.org/pmhp/ehippies/files/op1.pdf.

[Ins00] SANS Institute. Egress filtering v 0.2, 2000. Available at http://www.sans.org/y2k/egress.htm.

[KMC+00] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[Net00] Netscape. Leading Web sites under attack, February 2000. Available at http://technews.netscape.com/news/0-1007-200-1545348.html.

[Pac00] Packetstorm. Packetstorm, 2000. Available at http://packetstorm.securify.com.

[Pax97] Vern Paxson. End-to-End Routing Behavior in the Internet. *IEEE/ACM Transactions on Networking*, Vol.5, No.5:601–615, October 1997.

[Pax99] Vern Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks*, 31(23–24):2435–2463, December 1999.

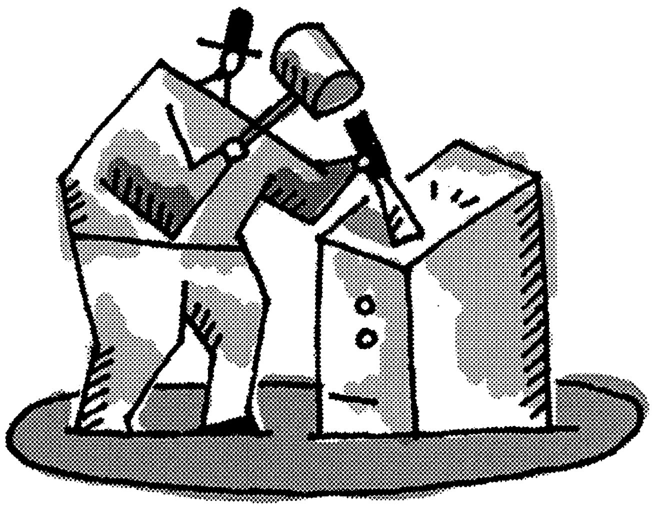
[SP01] Dawn Xiaodong Song and Adrian Perrig. Advanced and Authenticated Marking Schemes for IP Traceback. In *Proceedings of the IEEE Infocom 2001*, April 2001.

[Spi00] Lance Spitzner. The Tools and Methodologies of the Script Kiddie. Know Your Enemy, 2000. Available at `http://www.enteract.com/~lspitz/enemy.html`.

[Sto99] Robert Stone. CenterTrack: An IP Overlay Network for Tracking DoS Floods, October 1999. NANOG 17, Montreal, Canada.

[SWKA00] Stefan Savage, David Wetherall, Anna Karlin, and Tom Anderson. Practical Network Support for IP Traceback. In *Proceedings of the 2000 ACM SIGCOMM Conference*, pages 295–306, August 2000.

Hardware



Session Chair: Dirk Balfanz, *Xerox PARC*

Data Remanence in Semiconductor Devices

Peter Gutmann
IBM T.J.Watson Research Center
pgut001@cs.auckland.ac.nz

Abstract

A paper published in 1996 examined the problems involved in truly deleting data from magnetic storage media and also made a mention of the fact that similar problems affect data held in semiconductor memory. This work extends the brief coverage of this area given in the earlier paper by providing the technical background information necessary to understand remanence issues in semiconductor devices. Data remanence problems affect not only obvious areas such as RAM and non-volatile memory cells but can also occur in other areas of the device through hot-carrier effects (which change the characteristics of the semiconductors in the device), electromigration (which physically alter the device itself), and various other effects which are examined alongside the more obvious memory-cell remanence problems. The paper concludes with some design and device usage guidelines which can be useful in reducing remanence effects.

1. Introduction to Semiconductor Physics

Electrons surrounding an atomic nucleus have certain well-defined energy levels. When numbers of atoms are grouped together, the energy levels fall into certain fixed bands made up of the discrete energy levels of individual electrons. Between the bands are empty band gaps in which no electrons are to be found. A band which is completely empty or full of electrons cannot conduct (for an electron to move it must move to a higher discrete energy state, but in a completely full band this can't happen so a completely full band can conduct just as little as a completely empty one). An electron which is partaking in conduction is said to be in the conduction band, which lies immediately above the valence band.

At very low temperatures, the valence band for a semiconductor is full and the conduction band is empty, so that the semiconductor behaves like an insulator. As energy is applied, electrons move across the band gap from the valence band into the conduction band, leaving behind a hole which behaves like a positive charge carrier equal in magnitude to that of the electron as shown in Figure 1. Both the conduction and valence bands can conduct (via electrons or holes), producing a bipolar (two-carrier) conductor. In insulators the band gap is large enough that no promotion of electrons can occur. Conversely, conductors have conduction and valence bands which touch or even overlap.

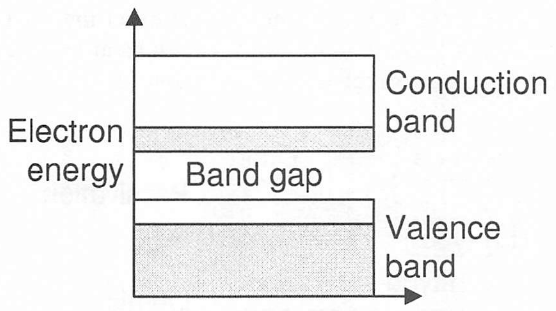


Figure 1: Electron behaviour in semiconductors

In order to make use of a semiconductor, we need to be able to produce material which carries current either through electrons or through holes, but not both. This is done by introducing impurities (usually called dopants) into the semiconductor lattice. For example adding boron (with three valence electrons) to silicon (with four valence electrons) leaves a deficiency of one electron per added boron atom, which is the same as one hole per boron atom. Conversely, adding phosphorus (with five valence electrons) leaves a surplus of one electron. Material doped to conduct mostly by holes is referred to as p-type; material doped to conduct mostly by electrons is called n-type.

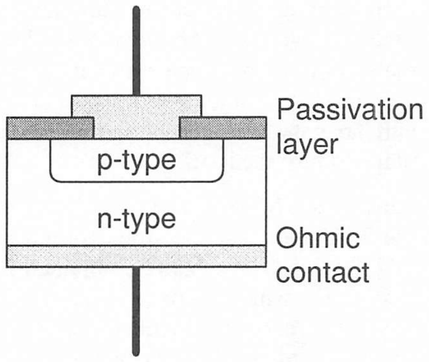


Figure 2: P-N junction diode

The makeup of a simple semiconductor device, the P-N junction diode, is illustrated in Figure 2. This consists of an n-type substrate with a p-type layer implanted into it. Protecting the surface is a thermally-grown oxide layer which serves to passivate and protect the silicon (this passivation layer is sometimes referred to as a tamperproof coating in smart card vendor literature). The p-type layer is formed by diffusing a dopant into the substrate at high temperatures through a hole etched into the passivation layer, or through ion-implantation.

When such a device is forward biased (a positive voltage applied to the p-type layer and a negative voltage applied to the n-type layer), current flows through the device. When the device is reverse-biased, very little current flows (at least until the device breakdown voltage is reached). The exact mechanism involved is fairly complex, further details are available from any standard reference on the topic [1].

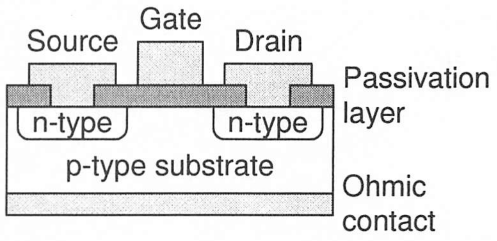


Figure 3: n-channel MOSFET

The semiconductor device used in almost all memories and in the majority of VLSI devices is the field-effect transistor (FET), specifically the metal oxide semiconductor FET (MOSFET). The structure of an n-channel MOSFET, a standard building block of semiconductor memories, is shown in Figure 3. When a voltage is applied to the gate, a conducting electron inversion layer is formed underneath it, giving this particular device the name of n-channel MOSFET. The n-type regions at the source and drain serve to supply electrons to the inversion layer during its formation, and the inversion layer, once formed, serves to connect the source and drain. Increasing the gate voltage increases the charge on the inversion layer and therefore the source-drain current. Enhancement-mode devices work in this manner, depletion-mode devices conduct with no gate voltage applied and require an applied voltage to turn them off.

Current flow in MOSFETs is dominated by electron/hole drift, and since electrons are more mobile than holes the fastest devices can be obtained by using n-channel devices which move electrons around. Because there are certain circuit advantages to be gained from combining n- and p-channel variants, many circuits use both in the form of complementary MOS (CMOS). Again, more details can be found in any standard reference [2].

2. Semiconductor Memories

Having covered the basic building blocks used to create memories, we can now go into the makeup of the memory devices themselves. In practice we distinguish between two main memory types, static RAM (SRAM) in which information is stored by setting the state of a bistable flip-flop which remains in this state as long as power is applied and no new data are written, and dynamic RAM (DRAM) in which information is stored by charging a capacitor which must be refreshed periodically as the charge bleeds away (a later section will cover EEPROM-based non-volatile memories). Because of their more complex circuitry, SRAMs typically only allow 25% of the density of DRAMs, but are sometimes preferred for their faster access times and low-power operation [3].

2.1. SRAM

SRAM cells are typically made up of cross-coupled inverters using the structure shown in Figure 4. The load devices can be polysilicon load resistors in older R-load cells, enhancement or depletion mode MOSFETs in an NMOS cell, or PMOS MOSFETs in a CMOS cell (providing an example of the previously mentioned combination of n-and p-channel MOSFET parts in a circuit). The purpose of the load devices is to offset the charge leakage at the drains of the data storage and cell selection MOSFETs. When the load is implemented with PMOS MOSFETs, the resulting CMOS cell has virtually no current flowing through it except during switching, leading to a very low power consumption.

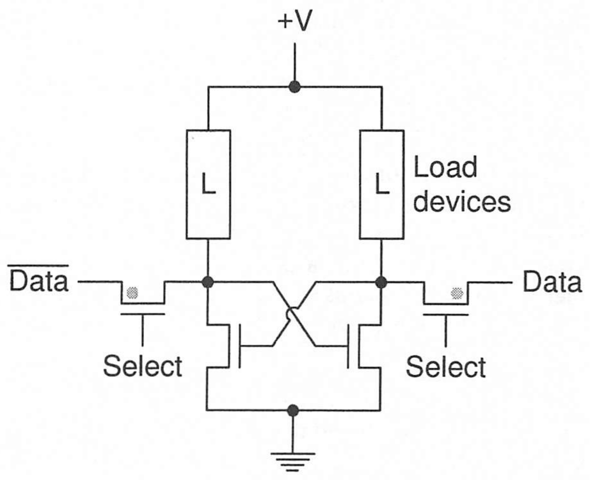


Figure 4: SRAM memory cell

Operation of the cell is very simple: When the cell is selected, the value written via Data/$\overline{\text{Data}}$ is stored in the cross-coupled flip-flops. The cells are arranged in an $n \times m$ matrix, with each cell individually addressable. Most SRAMs select an entire row of cells at a time, and read out the contents of all the cells in the row along the column lines.

2.2. DRAM

DRAM cells are made up of some device performing the function of a capacitor and transistors which are used to read/write/refresh the charge in the capacitors. Early designs used three-transistor (3T) cells, newer ones use a one-transistor (1T) cell as shown in Figure 5. Data is stored in the cell by setting the data line to a high or low voltage level when the select line is activated. Compare the simplicity of this circuit to the six-transistor SRAM cell!

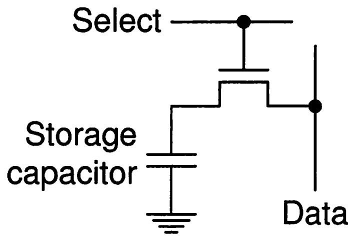


Figure 5: DRAM memory cell

The tricky parts of a DRAM cell lie in the design of the circuitry to read out the stored value and the design of the capacitor to maximise the stored charge/minimise the storage capacitor size. Stored values in DRAM cells are read out using sense amplifiers, which are extremely sensitive comparators which compare the value stored in the DRAM cell with that of a reference cell. The reference cell used is a dummy cell which stores a voltage halfway between the two voltage levels used in the memory cell (experimental multilevel cells use slightly different technology which won't be considered here). Later improvements in sense amplifiers reduced sensitivity to noise and compensated for differences in threshold voltages among devices.

3. DRAM Cell Structure

As has already been mentioned, the second tricky part of DRAM cell design is the design of the cell's storage capacitor. This typically consists of the underlying semiconductor serving as one plate, separated from the other polysilicon plate by a thin oxide film. This fairly straightforward two-dimensional cell capacitor was used in planar DRAM cells covering the range from 16 kb to 1 Mb cells, and placed the capacitor next to the transistor, occupying about a third of the total cell area. Although some gains in capacitance (leading to a shrinking of cell area) could be made by thinning the oxide thickness separating the capacitor plates, for newer cells it was necessary to move from the 2D plate capacitor structure to 3D structures such as trench and stacked capacitors. The conventional storage time (meaning the time during which the cell contents can be recovered without access to specialised equipment, typically 2-4 seconds [4]) for the memory cell is based on storage capacity and therefore the physical dimensions of the capacitor [5], so that DRAM designers have used various ingenious tricks to keep the capacitor storage constant while continuously shrinking cell dimensions.

Most of the earlier 4 Mb cells used trench capacitors, which had the advantage that capacitance could be increased by deepening the trench, which didn't use up any extra surface area. Newer generations of trench capacitor cells (sometimes called inverted trench cells) placed the storage electrode inside the trench, which reduced various problems encountered with the earlier cells which had the storage electrode in the substrate. There are a large number of variations possible with this cell, all of them based around the best way to implement the trench capacitor, with some relevant examples shown in Figure 6. The final evolution of the trench cell stacked the transistor above the capacitor, reducing the total area still further at the cost of increasing the number of steps required in the manufacturing process.

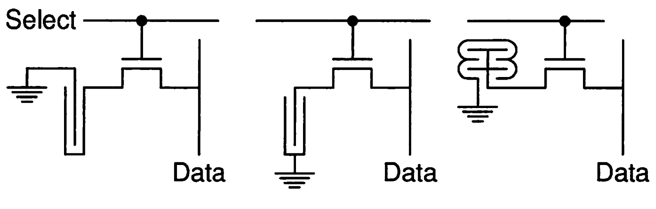


Figure 6: DRAM cells: Trench (left), inverted trench (middle), stacked (right)

Newer DRAM cells of 16 Mb and higher capacity moved from a menagerie of trench capacitor types to stacked capacitor cells (STCs), which stack the storage capacitor above the transistor rather than burying it in the silicon underneath. STCs used varying types of horizontal or vertical fins to further increase the surface area, and thus the capacitance. The cell at the right of Figure 6 employs a double-stacked STC. Another alternative to fins is spread-stacking, in which capacitors for different cells are layered over one another. As with trench capacitors, many further capacitor design variants exist [6][7].

4. Factors Influencing RAM Cell and General Device Operation

Now that we've covered the makeup of the various memory cell types, we can look at what makes it possible to analyse and recover data from these cells and from semiconductor devices in general long after it should (in theory) have vanished. To see how this is possible, we need to go back to the level of semiconductor device physics. Recall the discussion of (theoretical) electron/hole flow, in which electrons or holes move freely through a semiconductor lattice. In practice it isn't nearly this simple, since the lattice will contain impurities, atoms missing from the lattice (vacancies), and extra atoms in the lattice (interstitials). In addition, the atoms in the lattice will be vibrating slightly, producing phonons which work like electrons but carry momentum and can affect electrons if they collide with them.

If perchance these various impediments to free hole/electron movement don't take effect, or because of other factors such as high temperatures or voltages, electrons can build up quite a bit of momentum, which can be transferred to atoms in the lattice during collisions. In some cases this is enough to physically move the atom to new locations, a process known as electromigration.

4.1. Electromigration

Electromigration involves the relocation of metal atoms due to high current densities, a phenomenon in which atoms are carried along by an "electron wind" in the opposite direction to the conventional current flow, producing voids at the negative electrode and hillocks and whiskers at the positive electrode (if there's a passivation layer present the excess matter extrudes out to form a whisker, if not it distributes itself to minimise total surface area and forms a hillock). Void formation leads to a local increase in current density and Joule heating (the interaction of electrons and metal ions to produce thermal energy), producing further electromigration effects. When the external stress is removed, the disturbed system tends to relax back to its original equilibrium state, resulting in a backflow which heals some of the electromigration damage. In the long term though this can cause device failure (the excavated voids lead to open circuits, the grown whiskers to short circuits), but in less extreme cases simply serves to alter a device's operating characteristics in noticeable ways. For example the excavations of voids leads to increased wiring resistance, and the growth of whiskers leads to contact formation and current leakage. An example of a conductor which exhibits whisker growth due to electromigration is shown in Figure 7, and one which exhibits void formation (in this case severe enough to have lead to complete failure) is shown in Figure 8. Electromigration is a complex topic, an excellent introduction to the subject is contained in the survey paper by Lloyd [8].

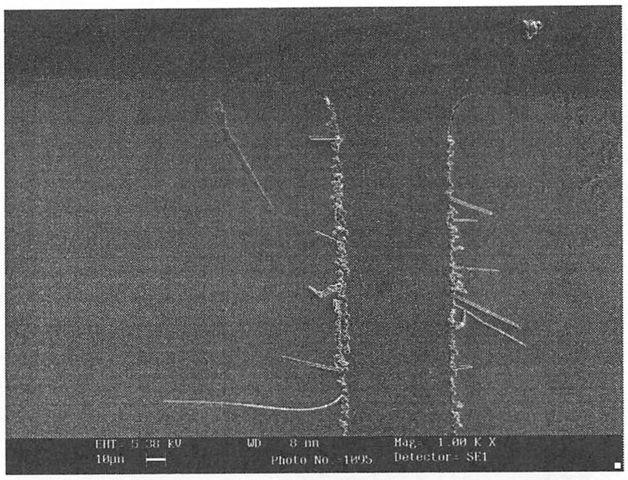


Figure 7: Whisker growth on a conductor due to electromigration

In order to reduce electromigration effects which occur in pure metals, interconnects are typically alloys (a few percent copper in aluminium interconnects, a few percent tin in copper interconnects) which have electromigration characteristics of their own in that the Cu or Sn solute atoms are displaced by the electron wind until the source region becomes depleted and behaves like the original pure metal. This initial level of electromigration effect, which doesn't affect circuit operation and isn't directly visible, can be detected using electron microprobe techniques which measure the distribution of the Cu or Sn along the base metal line [9].

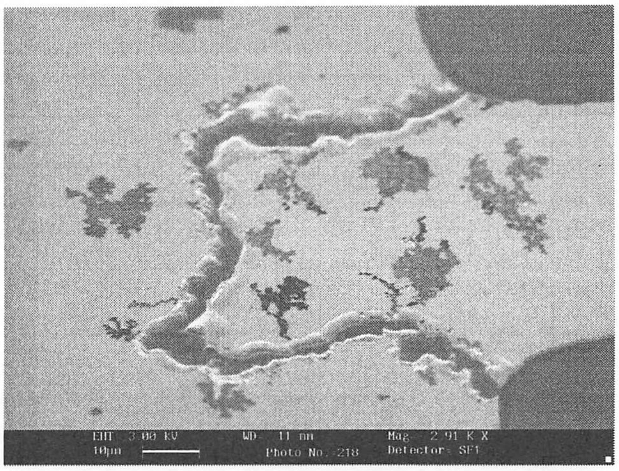


Figure 8: Void formation in a conductor due to electromigration

Although recent trends in clock speeds and device feature size reduction are resulting in devices with characteristics such as thin, sub-1.0µm lines, short sub-50-100µm line lengths, and utilisation of high frequencies which have traditionally been regarded as electromigration-resistant [10][11], they merely provide an ameliorative effect which is balanced by other (in some cases yet-to-be-understood) electromigration phenomena which occur as device dimensions shrink. Even the move to copper interconnects is no panacea, since although the actual copper electromigration mechanisms differ somewhat from those in aluminium, the problem still occurs [9][12][13].

4.2. Hot Carriers

High-energy electrons can cause other problems as well. A very obvious one is that the device heats up during operation because of collisions with the atoms in the lattice, at least one effect of the heating being the generation of further high-speed electrons. A problem which is particularly acute in MOSFETs with very small device dimensions is that of hot carriers which are accelerated to a high energy due to the large electric fields which occur as device dimensions are reduced (hot-carrier effects in newer high-density DRAMs have become so problematic that the devices contain internal voltage converters to reduce the external 3.3 or 5V supply by one or two volts to help combat this problem, and the most recent ones use a supply voltage of 2.5V for similar reasons). In extreme cases these hot electrons can overcome the Si-SiO_2 potential barrier and be accelerated into the gate oxide and stay there as excess charge [14]. The detrapping time for the resulting trapped charge can range from nanoseconds to days [15], although if the charge makes it into the silicon nitride passivation layer it's effectively there permanently (one study estimated a lifetime in excess of 30 years at 150°C) [16].

This excess charge changes the characteristics of the device over time, reducing the on-state current in n-MOSFETs and increasing the off-state current in p-MOSFETs [17][18][19]. The change in characteristics produces a variety of measurable effects, for example one study found a change of several hundred millivolts in memory cell signal voltage over a period of a few minutes [20]. This effect is most marked when a 1 bit is written after a 0 bit has been repeatedly read or written from the cell, leading to a drop in the cell threshold voltage. Writing a 0 over a 1 leads to an increase in the cell voltage. One way to detect these voltage shifts is to adjust the settings of the reference cell in the sense amplifier so that instead of being set to a median value appropriate for determining whether a stored value represents a 0 or a 1, it can be used to obtain a precise measurement of the actual voltage from the cell.

Hot-carrier stressing of cells can also affect other cell parameters such as the cell's access and refresh times. For example the precharge time (the time in which it takes to set the DRAM data lines to their preset values before an access) is increased by hot-carrier degradation, although the specific case of precharge time change affects only older NMOS cells and not newer CMOS ones. In addition hot carriers can produce visible or near-infrared photon emission in saturated FETs [21][22], but use of this phenomenon would require that an attacker be physically present while the device is being operated.

Hot carrier effects occur in logic circuits in general and not just in RAM cells. When MOS transistors are employed in digital logic, the logic steady states are regions of low stress because there is either a high field near the drain but the gate is low and the channel is off, or the electric field near the drain is low, in both cases leading to no generation of hot carriers. Hot carriers are generated almost exclusively during switching transitions [23][24]. The effects of the hot-carrier stressing can be determined by measuring a variety of device parameters, including assorted currents, voltages, and capacitances for the device [25].

4.3. Ionic Contamination

The most common ionic contamination present in semiconductors arises from the sodium (and to a lesser extent potassium) ions present in materials used during the semiconductor manufacturing and packaging process, a typical ion count being $10^{10}/cm^{2.}$ This contamination was originally thought to arise from sodium diffusion from the furnace tube [26] but with current manufacturing processes comes about because of impurities in the metallisation layers contaminating the silicon beneath. The problem is generally addressed through the standard application of passivation layers to protect the silicon [27]. Sodium ions have a fairly high mobility in silicon dioxide, and in the presence of an electric field or elevated temperatures will migrate towards the silicon/silicon dioxide interface in the device, reducing the threshold voltage of n-channel devices and increasing it for p-channel devices [28], again producing results which are detectable using the techniques described for hot-carrier effect detection.

There has been almost no work done in this area, probably because it isn't a significant enough problem to affect normal device operation, although one of the few works in this area indicate that it would take many minutes to hours of stress at standard operating temperatures (50-100°C) to produce any noticeable effect [26]. In addition it's unlikely that the effects of sodium contamination in current devices will be useful in recovering data from them, since reliability studies of devices indicate that contamination occurs only in randomly-distributed locations where impurities have penetrated the passivation layer through microfractures or pinholes [29]. Finally, the combination of improved manufacturing and passivation processes and shrinking device dimensions (which reduce the effects of mobile ions on the device) render this an area which is unlikely to bear much fruit.

Halide ions are another type of contaminant which may be introduced during the manufacturing process (in some cases deliberately as a semiconductor dopant), however these only lead to general corrosion of the device rather than producing any effects useful for recovering data from it (yet another reason why passivation layers are used is to provide some level of protection against this type of contamination and its attendant side-effects).

4.4. Other Effects

The storage capacitor in a DRAM cell typically needs to store 250-300 fC of charge. As has already been mentioned, earlier planar cells were scaled down by reducing the oxide thickness in the planar capacitor, while newer cells have gone to 3D structures such as trench and stacked capacitors. Trench capacitors typically used silicon dioxide (often referred to as ONO) insulators, while STCs have gone to using silicon nitride films which have a higher dielectric constant and allow thinner films to be used (as usual, a variety of other exotic technologies are also in use). In both cases parameters such as leakage current and time-dependant dielectric breakdown (TDDB) are relatively static and can't be used for stored data recovery purposes.

Radiation can also affect the operation of a RAM cell, for example radiation-induced charging of a MOSFET's gate oxide can alter the turn-on voltage of the device, with the oxide-trapped charge shifting the required turn-on voltage at the gate downwards for an n-channel MOSFET, effectively making it easier to turn on. p-channel MOSFETs, because of their slightly different mode of operation, are more resistant to radiation, but when affected become more difficult to turn off. Radiation can therefore alter memory cell parameters such as voltage level thresholds, timings, and power supply and leakage currents. As with DRAM capacitor effects this provides little practical help with stored data recovery, although it can be used to modify the operation of circuits for active attacks — as the radiation level increases it leads to losses in switching speed, a so-called "logic failure" in which a change in logic state becomes impossible. One way to utilise this in an attack would be to irradiate a cell until any erase-on-tamper functionality is rendered unusable, which is why high-end tamper-responsive crypto devices include sensors to detect the presence of ionising radiation [30].

A final problem area which is familiar to anyone who has examined the problems of erasing data stored on magnetic media is the fact that some of the more sophisticated memory designs include facilities to map out failing or failed cells in the same way that hard drives will map out bad sectors. This is performed using spare row/column line substitution (SLS), which substitutes problem cells with spare, redundant ones [31]. This technology is fairly rare and is usually applied only to correct initial hard failures so it isn't really a major concern, however it does become a problem in EEPROM/flash storage which is examined in Section 6.

4.5. Methods for Determining Changes in Device Operation

The techniques covered in the literature for determining changes in device operation are many and varied, which is both a blessing because there are so many to choose from and a curse because no two authors can agree on which criteria to use to determine a change in a device's operation, although there is general agreement that a device's characteristics have been altered once it has experienced a 100 mV shift in the device threshold voltage or a 10% change in transconductance, voltage, or current (depending on the author's preferences). Similarly, published results on phenomena such as hot-carrier effects are often obtained with specially-constructed test structures (ring oscillators are popular) which may not apply to other circuits such as memory cells. Because of the wide variation in experimental methods and sources reported in the literature and the equally large variety of devices in use it's not possible to provide definitive information on how the data recovery process might proceed, this section will attempt to cover some of the more common methods used for determining changes in device operation but is by no means exhaustive.

In the most extreme cases it may be possible to recover data directly from the device without resorting to any special techniques. "Burn-in" of data which had been stored in SRAM over long periods of time was common in 1980's devices, in one reported case DES master keys stored in a hardware security module used for PIN-processing were recovered almost intact on power-up, with recoverability of the remaining bits being aided by the presence of the DES key parity bits [32].

More recent SRAM devices are less likely to exhibit this problem to such a degree, requiring the use of more sophisticated readout methods. One widely-used technique from the field of device testing involves examining the amount of power supply current being supplied to the device, known as I_{DDQ} testing. The testing methodology involves executing a set of test vectors until a given location is reached (know as a parametric measurement stopping place or PM stop), at which point the device is halted and the current measured. In the quiescent state, n- and p-channel MOSFETS are either on or off, so there should be no current flowing, and PM stops are selected to coincide with this. Devices which aren't functioning normally will exhibit abnormal I_{DDQ} characteristics which can be measured once the PM stop is reached. By varying parameters such as the applied voltage and operating temperature, it's possible to identify devices which have been subject to effects such as hot-carrier stressing which have altered their operational parameters. Floating-gate designs may also have time-dependant I_{DDQ} characteristics in which the floating gate causes both n- and p-channel MOSFETS to be partially on and therefore conducting, a current flow which slowly ceases as the floating gate charges to a logic state and the current subsides. Again, the initial charge (or lack thereof) on the gate and the change in charge can be observed by observing the I_{DDQ} [33][34].

Many alternative techniques, arising from the field of semiconductor reliability analysis, also exist [35]. For example measuring the substrate current, the gate current, and the current in the gated drain-substrate diode of a MOSFET can all be used to determine the amount of stressing which has taken place [36][37]. These measurements can be used to determine the level and duration of stress applied [38].

Access to internal portions of a device can be obtained in many ways [39]. Most current ICs employ design for test (DFT) methodologies which break the device up into more manageable blocks of circuitry and provide test access to each block. Other techniques such as bond pad probing can also be used to obtain access to portions of a device.

When it becomes necessary to go beyond the access points provided by the manufacturer, things get a bit more tricky. Traditionally, access to internal portions of IC circuitry has been performed with mechanical probing techniques using tungsten wire etched down to a tip radius of 0.1–0.2 µm. These probes provide gigahertz bandwidths with an effective loading capacitance as low as 100 fF and a load resistance of 1 MΩ or more.

The recent use of deep submicron designs has complicated mechanical probing, since the optical diffraction limit and small depth of focus of the optical microscopes used to position the probes has made it difficult to see and probe the deep submicron lines. In addition standard mechanical probing isn't able to access buried lines in devices with multiple metallisation layers. Both of these limits can be overcome through the use of focused ion beam (FIB) workstations, which can be used both to expose buried conductors and to deposit new, easily-accessible probe points on an existing device [40] (this technique was used by the Canadian reverse-engineering lab Chipworks to rebuild an ATMEL EEPROM from a crashed aircraft in order to recover data from it [41]). The top metal layers are typically broad power buses, so no serious harm is caused by FIB milling of small holes to access lower-layer conductors. The only potential problem is that the FIB process can cause local charging of the device surface, which is usually avoided by grounding all pins in the device and shielding surrounding areas with conducting tape, however the FIB-induced charging can still affect floating gates so it's a good idea to avoid performing FIB surgery in their general vicinity [42]. In addition some technologies such as trench and STC DRAM cells are naturally resistant to being accessed in this manner, although it's still possible to get to transistors indirectly connected with the cell, for example the ones in the sense amplifiers.

5. Minimising RAM Data Recoverability

The previous sections have shown a variety of ways in which stored data can leave traces of its existence behind. These include the effects of electrical stress on ionic contaminants and hot-carrier effects (which can be used to recover overwritten data or data from memory to which power has been removed), and electromigration effects (which can be used to determine, after indefinite time periods, which type of signal was most commonly carried by a particular part of a circuit). The latter would prove useful in recovering information such as the bit patterns of keys stored in special-purpose cryptographic devices — since the physical device is modified the bits can be recovered an arbitrary amount of time later even if the memory cells they were stored in have been successfully erased and trapped charges have bled away.

The solution to the first problem is to ensure that sensitive data is stored for as short a time as possible; the solution to the second problem is more difficult but in general involves ensuring that a multitude of signals are sent through circuits without any one signal predominating. These approaches are explained in more detail in the following two sections.

5.1. Avoiding Short-term Retention Effects

The best way to avoid short-term retention effects is to ensure that no memory cell holds a data value for more than a certain amount of time. Based on the figures given earlier, a few minutes of storage of a given value should be treated as an upper bound; storage for any larger amount of time will cause detectable effects in the memory cell, although it may take quite a while longer before these effects really become a problem. In a series of tests carried out on a sample of SRAM devices, changes in device threshold voltage, transconductance, and drain-source current were observed after 100–500 seconds of stress, leading to a corresponding change in SRAM access time and operating voltage [43]. As the SRAM cell in Figure 4 indicates, reads and writes of 0 and 1 bits stress different access transistors in the cell so that it's possible to determine whether a 0 or 1 was stored there by determining which transistor was stressed the most (the grey dots in the figure indicate the main stress locations). The change in cell behaviour can be determined by recording the cell access time, through voltage microprobing of the cell's transistors, or using some of the other techniques mentioned earlier. Similar tests have been performed on DRAMs, although in this case the emphasis was on stress effects on shared circuitry such as address buffers and sense amplifiers. While there were quite noticeable effects in all of these areas the study didn't examine the effect on individual storage cells [44].

If nothing is done, the device will eventually recover by itself, although this can take quite some time at normal room temperatures. One way to accelerate the recovery process is to expose the device to elevated temperatures, the read access times for the SRAM devices mentioned previously were found to recover after around 1 ½ hours at 75°C, 3 days at 50°C, nearly two months at 20°C, and approximately 3 years at 0°C. No recovery was observed for write access times, but given that determining this would involve writing to the cells of interest it's unlikely that this presents much of a threat.

The best practical way to avoid long-term storage effects is to periodically flip the stored bits as suggested in the 1996 paper [45] so that each cell never holds a value long enough for it to be "remembered". Although impractical for large amounts of data, this may be feasible for small amounts of sensitive data such as cryptovariables. For example consider an encryption key whose bits are flipped once a minute. The key flip state is held in `keyState`, initially set to 0, and access is protected though a mutex `keyMutex`. The code to flip and use the bits is shown in Figure 9.

```
while( TRUE )
 {                    acquire keyMutex;
 acquire keyMutex;    if( keyState == 1 )
 key ^= 1111…1111;     key ^= 1111…1111;
 keyState ^= 1;       encrypt/decrypt;
 release keyMutex;    if( keyState == 1 )
                       key ^= 1111…1111;
 sleep( 60 );         release keyMutex;
 }
```

Figure 9: Flipping (left) and using (right) in-memory cryptovariables

This can be implemented as a simple wrapper around an existing encryption function, and ensures that the same key bits are never stored in a RAM cell for more than a certain amount of time, in this case one minute. A rather simpler solution which doesn't require complex bit-flipping and tracking of cryptovariable state information involves moving the data around in memory occasionally and overwriting the original storage locations, again ensuring that data is never stored in a RAM cell for too long.

If the luxury of custom circuitry is available (for example in a specialised crypto processor or module), it may be possible to integrate this bit-flipping into the memory circuitry. At each DRAM refresh cycle, the complement of the read value is written. When data is read from the cell, it is XORed with the `keyState` variable which tracks the state of the data currently stored in the cells (for older 3T cells in which the output data were inverted compared to the input data, it would have been possible to achieve this bit-flipping effect automatically by omitting the data inversion which is normally required during a refresh cycle).

Since SRAMs don't have a DRAM-style refresh cycle, this type of circuit modification isn't really possible for them, so that performing bit-flipping in an SRAM would require the addition of DRAM-style refresh circuitry, negating most of the advantages of SRAM.

Mention should also be made of hybrid memory types which combine DRAM with a small amount of SRAM (usually acting as some form of cache or I/O buffer) to improve the average access speed of the DRAM. A common example of this is extended data out (EDO) DRAM, which places a D-type latch on the data line so that the next access cycle can be started as soon as the data has entered the latches. Since these latches are shared across the entire DRAM, there is little chance of any piece of data except the last one read before a long break in accesses to the DRAM remaining in them for more than an instant, and if this is really a concern they can be flushed with a read to an innocuous memory location. Synchronous DRAMs (SDRAMs), which parallel load a quantity of data into a shift register and then shift it out one bit at a time, have similar properties.

5.2. Avoiding Long-term Retention Effects

Long-term retention effects are most likely to occur when the same data is repeatedly fed through a circuit, an example being the repeated use of a private key in a crypto accelerator for large-integer maths. This is a phenomenon which only occurs in specialised hardware, since general-purpose processors are fed such a variety of data that none of it has much effect on the circuitry. In contrast a private key stored in tamper-resistant hardware and fed repeatedly through a crypto processor will lead to some circuits always carrying the same signals, leading to long-term hot-carrier degradation and electromigration effects.

The most common solution to this problem (and that of device protection in general), embedding the crypto device in a tamper-resistant or tamper-sensing package which zeroises the cryptovariables when tampering is detected, is of little help since it's currently not possible to quickly zeroise electromigration effects, at least not without resorting to chemical zeroisation means. One way of undoing the effects of electromigration (apart from hoping that the system will eventually relax back to its ground state) is to apply a reverse current which reverses the electromigration stress, effectively undoing the electromigration damage [46][47]. This technique is already used in some EEPROM/flash devices to reduce erase stress by applying a reverse-polarity pulse after an erase pulse [48].

A somewhat more complex and difficult-to-implement approach is to have the crypto processor process dummy data when it isn't working with real data and keys. A downside of this is that it requires that a crypto operation be interruptible once started (it's no good having to wait for a dummy RSA decrypt to complete each time you want to decrypt data), and leads to increased power consumption and decreased device lifetime. In addition, it assumes that the device isn't occupied at all times with handling real data, leaving no chance to process any dummy data.

Unfortunately alternating dummy and real data is complicated by the design of typical crypto devices. For example encryption hardware will typically contain multiple key registers from which the currently selected key is expanded into storage reserved for the scheduled key, which is then used to encrypt a block of data. This means that switching keys incurs the overhead of a key schedule (although many devices, particularly DES hardware, will do an on-the-fly key schedule which is effectively free in hardware). In addition, pipelined implementations of block ciphers are generally not interruptible, requiring completion of processing of the current block (and in some cases several more blocks to force the pipeline to be flushed) before a key change can take effect.

In order to economise on chip real estate (and therefore on device cost), virtually all real-world/non-research DES hardware implementations iterate a single round 16 times, with on-the-fly key scheduling. Non-DES iterated algorithms (as well as non-crypto algorithms such as MD5 and SHA-1) are also implemented by iterating one round rather than by unrolling the operation. These can (with a little redesign) be interrupted at any point in the encryption/decryption cycle and new data can be substituted. In addition the fact that a single round is reused with multiple sets of key bits means that there's a very mixed set of data patterns in use which minimises the effects of any one pattern.

The crypto cores of large-integer maths accelerators (for example RSA accelerators) are less vulnerable to long-term effects since they typically iterate a simple operation such as addition or bit shifting in a loop to achieve multiplication, exponentiation, or whatever else is required. For example a typical RSA accelerator [49] might consist of one of more 512- or 1024-bit adders and/or shift registers which are used to perform RSA encryption using a series of squaring and modular multiplication steps, with a 1024-bit multiplication being performed with 1024 additions. Since the operations reuse the basic add/shift circuitry with constantly-changing bit patterns, the problem of data retention in these parts of the circuit are greatly reduced. However, the iterated application of the same keying data exacerbates the retention problem in other parts of the circuit, since a single modular exponentiation can result in key components travelling over the same data paths thousands or even millions of times. The RSA accelerator mentioned above, and others like it, perform a 1 kb modular multiplication with 1k modular additions, and a modular exponentiation with 1k modular multiplications, for a total of 1M applications of the same cryptovariables per RSA operation, and potentially trillions of applications per day of operation in a loaded SSL server.

6. EEPROM Memory Cells

Flash memory and EEPROMs are closely related, with flash being simply an extension of EEPROM technology to allow higher densities in exchange for some loss in flexibility. All EEPROM/flash memory cells work in the same general manner and employ as storage element a MOS transistor with a floating gate into which electrons are tunnelled using a process known as Fowler-Nordheim tunnelling, a quantum-mechanical effect in which electrons tunnel through the energy barrier of a very thin dielectric such as silicon dioxide [50].

6.1. FLOTOX Cells

A typical older EEPROM technology is Intel's floating-gate tunnelling oxide (FLOTOX) technology, with a typical transistor structure shown in Figure 10. A cross-section of the device with the corresponding energy-band diagram is shown in Figure 11. To store a charge, the control gate's voltage is raised with the source and drain grounded, so that electrons tunnel through to the floating gate. To remove the charge, the process is reversed and the electrons tunnel back out. The stored charge changes the threshold of the MOS transistor which comprises the cell, typically by 3–3.5V for a 5V cell [51]. The change in the threshold depends on a number of factors including the programming time (the longer the time, the larger the change), temperature (the higher the temperature, the fewer the available hot electrons available to be injected), and the condition of the cell, which is covered in more detail further on.

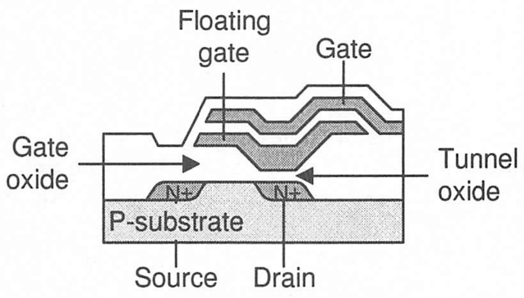


Figure 10: Typical EEPROM memory cell

This example of cell operation is merely representative, the details vary from manufacturer to manufacturer [52]. In particular, some issues like dielectric scaling effects and various program and erase mechanisms aren't fully understood yet, leading to a variety of technologies and continual changes in those technologies. In addition the interpretation of what represents a stored 0 or 1 varies from device to device in that cells can be written into either state, with one state being regarded as "programmed" and the other as "erased". In some cells the low-stored-charge state is called programmed, in others it's called erased.

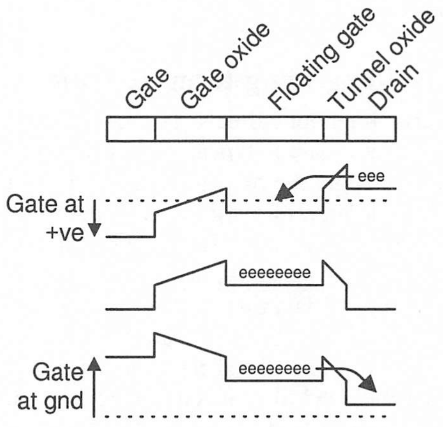


Figure 11: FLOTOX EEPROM program/erase process

6.2. ETOX Cells

A somewhat newer technology is represented by Intel's EPROM tunnel oxide (ETOX) cell [53][54], which uses channel hot electron (CHE) injection to store a value and Fowler-Nordheim tunnelling to remove it, is illustrated in Figure 12. This technique is widely used in flash memory, although the widely-used NAND flash again uses tunnelling for both programming and erasure (NAND flash cells have a somewhat specialised architecture which allows the use of the more efficient tunnelling for program and erase [55]).

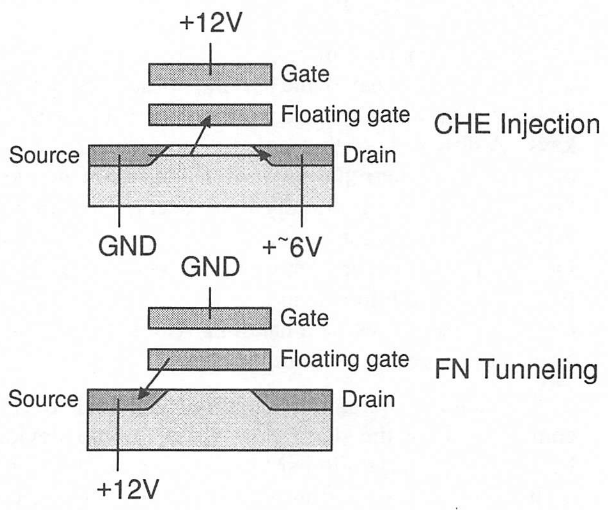


Figure 12: ETOX EEPROM program/erase process

The basic EEPROM cell consists of the storage transistor described above and a second transistor to select or deselect the cell (some technologies employ additional error detection and correction circuitry). In an attempt to increase storage density, manufacturers have moved towards using the select transistors to handle multiple storage cells. When the cells are organised in this manner only the programming step can be done in a bit-by-bit basis, the erase operation works by erasing all cells in a block and programming the new data bits as required (or rewriting the old data in sections where no change is to occur). Because programming is possible on a bit by bit basis, it's usual to only program cells which are currently in the erased state to avoid overprogramming already-programmed cells and (in the case of flash memory) to avoid having to erase an entire sector just to change one or two bytes.

The details of the erase operation again vary somewhat across different manufacturers, and unlike programming the erase operation functions on a block of cells at a time. Since the cells aren't all uniform, a cell array may contain fast-erasing bits as well as typical-erase bits, so that a single erase pulse may not erase all the cells. Because of this it's necessary to verify the erase and reapply the erase pulse to catch the remaining cells. This operation is repeated until all cells have been reduced to less than the cell erase verify level. In practice the erasure process is a speculative one, with the initial pulse being far shorter than the typical erase time, followed by longer and longer pulses as required. The reason for using this erase process is that we want to avoid further affecting already-erased cells, once a cell is erased by a pulse any subsequent pulses don't significantly change its threshold voltage. The programming process is usually performed using a similar type of algorithm, with the main difference being that programming is possible on a bit-by-bit basis so that cells which are already at the required level aren't programmed further [56][57].

6.3. Flash Memory Technology

The simplest flash technology, employing a NOR structure, allows access to individual cells but requires a dual-voltage supply and has a rather low block density. More common is a NAND structure in which multiple transistors in series are controlled by a single select transistor as shown in Figure 13. NAND EEPROM/flash moves data to and from storage in large blocks, typically 64–256 bytes at a time, and has cells which are typically one-quarter the size of equivalent conventional EEPROM cells. Other size optimisations include tricks such as stacking the select transistor atop the storage transistor and similar methods for merging the function of the two transistors into a single, smaller unit, for example including the select gate as a second gate in the cell, the sidewall select-gate or SISOS cell [58]. Another way to improve density is to use multilevel storage, which distinguishes between multiple charge levels in a cell instead of just the basic programmed and erased states [59][60].

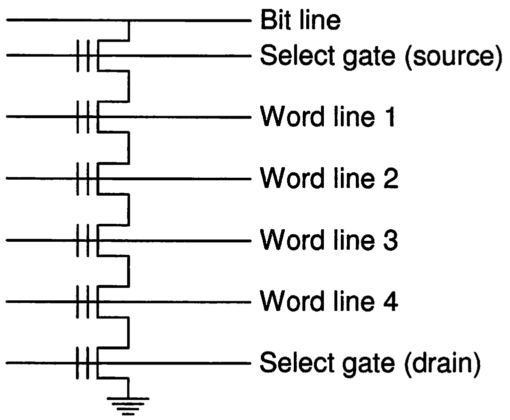


Figure 13: NAND flash memory structure

6.4. Data Remanence in EEPROM/Flash Memory

The number of write cycles possible with EEPROM technology is limited because the floating gate slowly accumulates electrons, causing a gradual increase in the storage transistor's threshold voltage which manifests (in its most observable form) as increased programming time and, eventually, an inability to erase the cell. Although EEPROM/flash cells can typically endure 1M or more write/erase cycles, the presence of slight defects in the tunnelling oxide (leading to leakage and eventual breakdown during the tunnelling process) reduces the effective life of the entire collection of cells to 10–100k write/erase cycles. This problem is significantly reduced in flash memory cells, where the main failure mode appears to be negative charge trapping (that is, the trapping of holes in the gate oxide) which inhibits further CHE injection and tunnelling, changing the write and erase times of the cell and shifting its threshold voltage [61][62]. The amount of trapped charge can be determined by measuring the gate-induced drain leakage (GIDL) current of the cell [63], or its effects can be observed more indirectly by measuring the threshold voltage of the cell. In older devices which tied the reference voltage used to read the cell to the device supply voltage it was often possible to do this (and perform other interesting tricks such as making a programmed cell appear erased and vice versa) by varying the device supply voltage, but with newer devices it's necessary to change the parameters of the reference cells used in the read process, either by re-wiring portions of the cell circuitry or by using undocumented test modes built into the device by manufacturers.

A less common failure mode which occurs with the very thin tunnel oxides used in flash memory is one where unselected erased cells adjacent to selected cells gain charge when the selected cell is written (known as a programming disturb) due to the gate of the unselected transistor being stressed by the voltage applied to the common data line shared with the selected transistor. There are various subfamilies of programming disturbs including bitline (also called drain-) and word line (also called gate-) disturbs, in which bias on the common bit or word line causes charge to be injected from the substrate into the floating gate of an unselected cell [64][65]. This isn't enough to change the cell threshold sufficiently to upset a normal read operation, but should be detectable using the specialised techniques described above. There is also a type of disturb which can occur when extensive read cycles are performed, with this type of disturb holes are generated in the substrate via impact ionisation and injected into the floating gate, causing a loss of charge. This appears to only affect so-called fast-programming cells [66] (which erase and program a lot quicker than typical cells) and isn't useful in determining the cell contents since it requires knowledge of the cell's pre-stress characteristics to provide a baseline to compare the post-stress performance to.

In terms of long-term retention issues, there is a phenomenon called field-assisted electron emission in which electrons in the floating gate migrate to the interface with the underlying oxide and from there tunnel into the substrate, causing a net charge loss. The opposite occurs with erased cells, in which electron injection takes place [67]. Finally, just as with DRAM cells, EEPROM/flash cells are also affected by ionic contamination since the negatively-charged floating gate attracts positive ions which induce charge loss, although the effect is generally too miniscule to be measurable.

The means of detecting these effects is as for RAM cells and MOSFET devices in general, for example measuring the change in cell threshold, gate voltage, or observing other phenomena which can be used to characterise the cell's operation. The changes are particularly apparent in virgin and freshly-programmed cells, where the first set of write/erase cycles causes a (comparatively) large shift in the cell thresholds, after which changes are much more gradual [52][65] (as usual, this is device-dependant, for example the high injection MOS or HIMOS cell exhibits somewhat different behaviour than FLOTOX and ETOX cells [68]). Because of this it's possible to differentiate between programmed-and-erased and never-programmed cells, particularly if the cells have only been programmed and erased once, since the virgin cell characteristics will differ from the erased cell characteristics. Another phenomenon which helps with this is overerasing, in which an erase cycle applied to an already-erased cell leaves the floating gate positively charged, thus turning the memory transistor into a depletion-mode transistor. To avoid this problem, some devices first program all cells before erasing them (for example Intel's original ETOX-based devices did this, programming the cells to 0s before erasing them to 1s [54]), although the problem is more generally solved by redesigning the cell to avoid excessive overerasing, however even with this protection there's still a noticeable threshold shift when a virgin cell is programmed and erased.

EEPROM/flash memory can also have its characteristics altered through hot carriers which are generated by band-to-band tunnelling and accelerated in the MOSFET's depletion region, resulting in changes in the threshold voltages of erased cells. As with other factors which affect EEPROM/flash cells, the changes are particularly apparent in fresh cells but tend to become less noticeable after around 10 program/erase cycles [61].

Finally, as with SLS features in RAM, EEPROM/flash memory often contains built-in features which allow the recovery of data long after it should have, in theory, been deleted. The mapping out of failing sectors which parallels the sector sparing used in disk drives has already been mentioned, there also exist device-specific peculiarities such as the fact that data can be recovered from the temporary buffers used in the program-without-erase mode employed in some high-density flash memories, allowing recovery of both the new data which was written and the original data in the sector being written to [60].

Working at a slightly higher level than the device itself are various filesystem-level wear-levelling techniques which are used to decrease the number of erase operations which are necessary to update data, and the number of writes to a single segment of flash [69]. Flash file systems are generally log-structured file systems which write changed data to a new location in memory and garbage-collect leftover data in the background or as needed, with the exact details being determined by a cleaning policy which determines which memory segments to clean, when to clean them, and where to write changed data [70][71][72]. Because of this type of operation it's not possible to cycle fresh cells to reduce remanence effects without bypassing the filesystem, in fact the operation of the wear-levelling system acts to create a worst-case situation in which data is always written to fresh cells. Trying to burn in an area of storage by creating a file and overwriting it 10-100 times will result in that many copies of the data being written to different storage locations, followed by the actual data being written to yet another fresh storage location. Even writing enough data to cycle through all storage locations (which may be unnecessarily painful when the goal is to secure a 1 kB data area on a device containing 256 MB of non-critical data) may not be sufficient, since pseudorandom storage location selection techniques can result in some locations being overwritten many times and others being overwritten only a handful of times.

There is no general solution to this problem, since the goal of wear-levelling is the exact opposite of the (controlled) wearing which is needed to avoid remanence problems. Some possible application-specific solutions could include using direct access to memory cells if available, or using knowledge of the particular device- or filesystem's cleaning policy to try and negate it and provide the required controlled wearing. Since this involves bypassing the primary intended function of the filesystem, it's a somewhat risky and tricky move.

7. Conclusion

Although the wide variety of devices and technologies in use, and the continuing introduction of new technologies not explicitly addressed in this work, make providing specific guidelines impossible, the following general design rules should help in making it harder to recover data from semiconductor memory and devices:

- Don't store cryptovariables for long time periods in RAM. Move them to new locations from time to time and zeroise the original storage, or flip the bits if that's feasible.
- Cycle EEPROM/flash cells 10-100 times with random data before writing anything sensitive to them to eliminate any noticeable effects arising from the use of fresh cells (but see also the point further down about over-intelligent non-volatile storage systems).
- Don't assume that a key held in RAM in a piece of crypto hardware such as an RSA accelerator, which reuses the same cryptovariable(s) constantly, has been destroyed when the RAM has been cleared. Hot-carrier and electromigration effects in the crypto circuitry could retain an afterimage of the key long after the original has leaked away into the substrate.
- As a corollary, try and design devices such as RSA accelerators which will reuse a cryptovariable over and over again in such a way that they avoid repeatedly running the same signals over dedicated data lines.
- Remember that some non-volatile memory devices are a little too intelligent, and may leave copies of sensitive data in mapped-out memory blocks after the active copy has been erased. Devices and/or filesystems which implement wear-levelling techniques are also problematic since there's no way to know where your data is really going unless you can access the device at a very low level.

Finally, however, the best defence against data remanence problems in semiconductor memory is, as with the related problem of data stored on magnetic media, the fact that ever-shrinking device dimensions (DRAM density is increasing by 50% per year [73]), and the use of novel techniques such as multilevel storage (which is being used in flash memory and may eventually make an appearance in DRAM as well [74]) is making it more and more difficult to recover data from devices. As the 1996 paper suggested for magnetic media, the easiest way to make the task of recovering data difficult is to use the newest, highest-density (and by extension most exotic) storage devices available.

Acknowledgements

The author would like to thank Steve Weingart and the referees for their feedback and comments on this paper and Dr.Veng-cheong Lo for permission to reproduce the electromigration images.

References

[1] "Introductory Semiconductor Device Physics", Greg Parker, Prentice Hall, 1994.

[2] "Fundamentals of Modern VLSI Devices", Yuan Taur and Tak Ning, Cambridge University Press, 1998.

[3] "Semiconductor Memories: Technology, Testing, and Reliability", Ashok Sharma, IEEE Press, 1997.

[4] "DRAM Variable Retention Time", P.Restle, J.Park, and B.Lloyd, International Electron Devices Meeting (IEDM'92) Technical Digest, December 1992, p.807.

[5] "A Numerical Analysis of the Storage Times of Dynamic Random-Access Memory Cells Incorporating Ultrathin Dielectrics", Alex Romanenko and W.Milton Gosney, *IEEE Transactions on Electron Devices*, **Vol.45, No.1** (January 1998), p.218.

[6] "Advanced Cell Structures for Dynamic RAMs", Nicky Lu, *IEEE Circuits and Devices Magazine*, **Vol.5, No.1** (January 1989), p.27.

[7] "DRAM Technology Perspective for Gigabit Era", Kinam Kim, Chang-Gyu Hwang, and Jong Gil Lee, *IEEE Transactions on Electron Devices*, **Vol.45, No.3** (March 1998), p.598.

[8] "Electromigration for Designers: An Introduction for the Non-Specialist", J.Lloyd,

`http://www.simplex.com/udsm/whitepapers/electromigration1/index.html.`

[9] "*In situ* scanning electron microscope comparison studies on electromigration of Cu and Cu(Sn) alloys for advanced chip interconnects", K.Lee, C.Hu, and K.Tu, *Journal of Applied Physics*, **Vol.78**, **No.7** (1 October 1995), p.4428.

[10] "Electromigration in Metals", Paul Ho and Thomas Kwok, *Reports on Progress in Physics*, **Vol.52**, **Part 1** (1989), p.301.

[11] "Theoretical and Experimental Study of Electromigration", Jian Zhao, "Electromigration and Electronic Device Degradation", John Wiley and Sons, 1994, p.167.

[12] "On the unusual electromigration behaviour of copper interconnects", E.Glickman and M.Nathan, *Journal of Applied Physics*, **Vol.80**, **No.7** (1 October 1996), p.3782.

[13] "Surface Electromigration in Copper Interconnects", N.McCusker, H.Gamble, and B.Armstrong, *Proceedings of the International Reliability Physics Symposium (IRPS 1999)*, March 1999, p.270.

[14] "Hot-Carrier Effects in MOS Devices", Eiji Takeda, Cary Yang, and Akemi Miura-Hamada, Academic Press, November 1995.

[15] "Dynamic Degradation in MOSFET's — Part I: The Physical Effects", Martin Brox and Werner Weber, *IEEE Transactions on Electron Devices*, **Vol.38**, **No.8** (August 1991), p.1852.

[16] "Data Storage in NOS: Lifetime and Carrier-to-Noise Measurements", Bruce Terris and Robert Barrett, *IEEE Transactions on Electron Devices*, **Vol.42**, **No.5** (May 1995), p.944.

[17] "How do Hot Carriers Degrade N-Channel MOSFETs?", Kaizad Mistry and Brian Doyle, *IEEE Circuits and Devices*, **Vol.11**, **No.1** (January 1995), p.28.

[18] "Hot-carrier Damage in AC-Stressed Deep Submicrometer CMOS Technologies", A.Bravaix, D.Doguenheim, N.Revil, and E.Vincent, IEEE Integrated Reliability Workshop (IRW'99) Final Report, October 1999, p.61.

[19] "Hot-carrier Degradation Evolution in Deep Submicrometer CMOS Technologies", A.Bravaix, IEEE Integrated Reliability Workshop (IRW'99) Final Report, October 1999, p.174.

[20] "Reduction of Signal Voltage of DRAM Cell Induced by Discharge of Trapped Charges in Nano-meter Thick Dual Dielectric Film", J.Kumagai, K.Toita, S.Kaki, and S.Sawada, *Proceedings of the International Reliability Physics Symposium (IRPS 1990)*, April 1990, p.170.

[21] "Extended (1.1-2.9eV) Hot-Carrier Induced Photon Emission in n-Channel MOSFETs", M.Lanzoni, E.Sangiorgi, C.Fiegna, and B.Riccò, International Electron Devices Meeting (IEDM'90) Technical Digest, December 1990, p.69.

[22] "Time-Resolved Optical Characterisation of Electrical Activity in Integrated Circuits", James Tsang, Jeffrey Kash, and David Vallett, *Proceedings of the IEEE*, **Vol.88**, **No.9** (September 2000), p.1440.

[23] "Setting the Trap for Hot Carriers", Shian Aur, Charvaka Duvvury, and William Hunter, *IEEE Circuits and Devices*, **Vol.11**, **No.4** (July 1995), p.18.

[24] "Design Considerations for CMOS Digital Circuits with Improved Hot-Carrier Reliability", Yusuf Leblebici, *IEEE Journal of Solid-state Circuits*, **Vol.31**, **No.7** (July 1996), p.1014.

[25] "Hot-Carrier-Induced Alterations of MOSFET Capacitances: A Quantitative Monitor for Electrical Degradation", David Esseni, Augusto Pieracci, Manrico Quadrelli, and Bruno Riccò, *IEEE Transactions on Electron Devices*, **Vol.45**, **No.11** (November 1998), p.2319.

[26] "Mobile ion effects in low-temperature silicon oxides", N.Young, A.Gill, and I.Clarence, *Journal of Applied Physics*, **Vol.66**, **No.1** (1 July 1989), p.187.

[27] "Built-in Reliability Through Sodium Elimination", Jeff Chinn, Yueh-Se Ho, and Mike Chang, *Proceedings of the International Reliability Physics Symposium (IRPS 1994)*, April 1994, p.249.

[28] "Failure Modes and Mechanisms for VLSI ICs — A Review", Fausto Fantini and Carlo Morandi, *IEE Proceedings — Part G: Electronic Circuits and Systems*, **Vol.132**, **No.3** (June 1985), p.74.

[29] "An Accelerated Sodium Resistance Test for IC Passivation Films", Charlie Hong, Brent Henson, Tony Scelsi, and Robert Hance, *Proceedings of the International Reliability Physics Symposium (IRPS 1996)*, May 1996, p.318.

[30] "Building a High-Performance, Programmable Secure Coprocessor", Sean Smith and Steve Weingart, *Computer Networks*, **Vol.31**, **No.4** (April 1999), p.831.

[31] "Memory LSI Reliability", Masao Fukuma, Hiroshi Furuta, and Masahide Takada, *Proceedings of the IEEE*, **Vol.81**, **No.5** (May 1993), p.768.

[32] "Low Cost Attacks on Tamper Resistant Devices", Ross Anderson and Markus Kuhn, *Proceedings of the 5th International Workshop on Security Protocols*, Springer-Verlag LNCS No.1361, April 1997.

[33] "IDDQ Testing for High Performance CMOS — The Next Ten Years", T.Williams, R.Kapur, M.Mercer, R.Dennard, and W.Maly, *Proceedings of the European Design and Test Conference (EDTC'96)*, 1996, p.578.

[34] "Electrical Characterization", Steven Frank, Wilson Tan, and John West, in "Failure Analysis of Integrated Circuits: Tools and Techniques", Kluwer Academic Publishers, 1999, p.13.

[35] "Semiconductor Material and Device Characterization (2nd Ed)", Dieter Schroder, John Wiley and Sons, 1998.

[36] "Characterization of Hot-Electron-Stressed MOSFET's by Low-Temperature Measurements of the Drain Tunnel Current", Alexandre, Acovic, Michel Dutoit, and Marc Ilegems, *IEEE Transactions on Electron Devices*, **Vol.37**, **No.6** (June 1990), p.1467.

[37] "Monitoring Trapped Charge Generation for Gate Oxide Under Stress", Yung Hao Lin, Chung Len Lee, and Tan Fu Lei, *IEEE Transactions on Electron Devices*, **Vol.44, No.9** (September 1997), p.1441.

[38] "Characteristic length and time in electromigration", Morris Shatzkes and Yusue Huang, *Journal of Applied Physics*, **Vol.74, No.11** (December 1993), p.6609.

[39] "IC Failure Analysis: Techniques and Tools for Quality and Reliability Improvement", Jerry Soden and Richard Anderson, *Proceedings of the IEEE*, **Vol.81, No.5** (May 1993), p.703.

[40] "The role of focused ion beams in physical failure analysis", G.Matusiewicz, S.Kirch, V.Seeley, and P.Blauner, *Proceedings of the International Reliability Physics Symposium (IRPS 1991)*, April 1991, p.167.

[41] "Chip Detectives", Jean Kumagai, *IEEE Spectrum*, **Vol.37, No.11** (November 2000), p.43.

[42] "Probing Technology for IC Diagnosis", Christopher Talbot, in "Failure Analysis of Integrated Circuits: Tools and Techniques", Kluwer Academic Publishers, 1999, p.113.

[43] "Relation between the hot carrier lifetime of transistors and CMOS SRAM products", Jacob van der Pol and Jan Koomen, *Proceedings of the International Reliability Physics Symposium (IRPS 1990)*, April 1990, p.178.

[44] "Hot-carrier-induced Circuit Degradation in Actual DRAM", Yoonjong Huh, Dooyoung Yang, Hyungsoon Shin, and Yungkwon Sung, *Proceedings of the International Reliability Physics Symposium (IRPS 1995)*, April 1995, p.72.

[45] "Secure Deletion of Data from Magnetic and Solid-State Memory", Peter Gutmann, *Proceedings of the 6th Usenix Security Symposium*, July 1996, p.77.

[46] "Metal Electromigration Damage Healing Under Bidirectional Current Stress", Jiang Tao, Nathan Cheung, and Chenming Ho, *IEEE Electron Device Letters*, **Vol.14, No.12** (December 1993), p.554.

[47] "An Electromigration Failure Model for Interconnects Under Pulsed and Bidirectional Current Stressing", Jiang Tao, Nathan Cheung, and Chenming Ho, *IEEE Transactions on Electron Devices*, **Vol.41, No.4** (April 1994), p.539.

[48] "New Write/Erase Operation Technology for Flash EEPROM Cells to Improve the Read Disturb Characteristics", Tetsuo Endoh, Hirohisa Iizuka, Riichirou Shirota, and Fujio Masuoka, *IEICE Transactions on Electron Devices*, **Vol.E80-C, No.10** (October 1997), p.1317.

[49] "A High-Speed RSA Encryption LSI with Low Power Dissipation", A.Satoh, Y.Kobayashi, H.Niijima, N.Ooba, S.Munetoh, and S.Sone, *Proceedings of the Information Security Workshop (ISW'97)*, Springer-Verlag LNCS No.1396, September 1997.

[50] "Nonvolatile Semiconductor Memory Technology: A Comprehensive Guide to Understanding and Using NVSM Devices", William Brown and Joe Brewer (eds), IEEE Press, 1998.

[51] "Hot-electron injection into the oxide in n-channel MOS devices", Boaz Eitan and Dov Frohman-Bentchkowsky, *IEEE Transactions on Electron Devices*, **Vol.28, No.3** (March 1981), p.328.

[52] "Analysis and Modeling of Floating-gate EEPROM Cells", Avinoam Kolodny, Sidney Nieh, and Boaz Eitan, *IEEE Transactions on Electron Devices*, **Vol.33, No.6** (June 1986), p.835

[53] "An In-System Reprogrammable 256K CMOS Flash Memory", Virgil Kynett, Alan Baker, Mickey Fandrich, George Hoekstra, Owen Jungroth, Jerry Kreifels, and Steven Wells, *Proceedings of the IEEE International Solid State Circuits Conference*, February 1988, p.132.

[54] "An In-System Reprogrammable 32K×8 CMOS Flash Memory", Virgil Kynett, Alan Baker, Mick Fandrich, George Hoekstra, Owen Jungroth, Jerry Kreifels, Steven Wells, and Mark Winston, *IEEE Journal of Solid-state Circuits*, **Vol.23, No.5** (October 1988), p.1157.

[55] "New Ultra High Density EPROM and Flash EPROM Cell with NAND Structure", Fujio Masuoka, Masaki Momodomi, Yoshihisa Iwata, and Riichirou Shirota, International Electron Devices Meeting (IEDM'87) Technical Digest, 1987, p.552.

[56] "A 4-Mb NAND EEPROM with Tight Programmed V_t Distribution", Masaki Momodomi, Tomoharu Tanaka, Yoshihisa Iwata, Yoshiyuki Tanaka, Hideko Oodaira, Yasuo Itoh, Riichirou Shirota, Kazunori Ohuchi, and Fujio Masuoka, *IEEE Journal of Solid-state Circuits*, **Vol.26, No.4** (April 1991), p.492.

[57] "A Quick Intelligent Page-Programming Architecture and a Shielded Bitline Sensing Method for 3V-Only NAND Flash Memory", Tomoharu Tanaka, Yoshiyuki Tanaka, Hiroshi Nakamura, Koji Sakui, Hideko Oodaira, Riichirou Shirota, Kazunori Ohuchi, Fujio Masuoka, and Hisashi Hara, *IEEE Journal of Solid-state Circuits*, **Vol.29, No.11** (November 1994), p.1366.

[58] "Flash Memory Cells — An Overview", Paolo Pavan, Roberto Bez, Piero Olivo, and Enrico Zanoni, *Proceedings of the IEEE*, **Vol.85, No.8** (August 1997), p.1248.

[59] "A multilevel-cell 32Mb flash memory", M.Bauer, R.Alexis, B.Atwood, K.Fazio, K.Frary, M.Hensel, M.Ishac, J.Javanifard, M.Landgraf, D.Leak, K.Loe, D.Mills, P.Ruby, R.Rozman, S.Sweha, K.Talreja, and K.Wojciechowski, *Proceedings of the IEEE International Solid State Circuits Conference*, February 1995, p.132.

[60] "A 256-Mb Multilevel Flash Memory with 2-MB/s Program Rate for Mass Storage Applications", Atsushi Nozoe, Hiroaki Kotani, Tetsuya Tsujikawa, Keiichi Yoshida, Kazunori Furusawa, Masataka Kato, Toshiaki Nishimoto, Hitoshi Kume, Hideaki Kurata, Naoki Miyamoto, Shoji Kubono, Michitaro Kanamitsu, Kenji Koda, Takeshi Nakayama, Yasuhiro Kouro, Akira Hosogane, Natsuo Ajika, and Kiyoteru Kobayashi, *IEEE Journal of Solid-state Circuits*, **Vol.34, No.11** (November 1999), p.1544.

[61] "Degradations due to Hole Trapping in Flash Memory Cells", Sameer Haddad, Chi Chang, Balaji Swaminathan, and Jih Lien, *IEEE Electron Device Letters*, **Vol.10**, **No.3** (March 1989), p.117.

[62] "Degradation of Tunnel-Oxide Floating-Gate EEPROM Devices and the Correlation with High Field-Current-Induced Degradation of Thin Gate Oxides", Johan Witters, Guido Groeseneken, and Herman Maes, *IEEE Transactions on Electron Devices*, **Vol.36**, **No.9** (September 1989), p.1663.

[63] "Determination of Trapped Oxide Charge in Flash EPROMs and MOSFETs with Thin Oxides", K.Tamer San and Tso-Ping Ma, *IEEE Electron Device Letters*, **Vol.13**, **No.8** (August 1992), p.439.

[64] "Reliability Issues of Flash Memory Cells", Seiichi Aritome, Riichiro Shirota, Gertjan Hemink, Tetsuo Endoh, and Fujio Masuoka, *Proceedings of the IEEE*, **Vol.81**, **No.5** (May 1993), p.776.

[65] "Effects of Erase Source Bias on Flash EPROM Device Reliability", K.Tamer San, Çetin Kaya, and T.P.Ma, *IEEE Transactions on Electron Devices*, **Vol.42**, **No.1** (January 1995), p.150.

[66] "Flash EPROM Disturb Mechanisms", Clyde Dunn, Çetin Kaya, Terry Lewis, Tim Strauss, John Schreck, Pat Hefly, Matt Middendorf, and Tamer San, *Proceedings of the International Reliability Physics Symposium (IRPS 1994)*, April 1994, p.299.

[67] "Retention characteristics of single-poly EEPROM cells", C.Papadas, G.Ghibaudo, G.Pananakakis, C.Riva, P.Ghezzi, C.Gounelle, and P.Mortini, *Proceedings of the European Symposium on Reliability of Electron Devices, Failure Physics and Analysis*, October 1991, p.517.

[68] "Write/Erase Degradation in Source Side Injection Flash EEPROMs: Characterization Techniques and Wearout Mechanisms", Dirk Wellekens, Jan Van Houdt, Lorenzo Faraone, Guido Groeseneken, and Herman Maes, *IEEE Transactions on Electron Devices*, **Vol.42**, **No.11** (November 1995), p.1992.

[69] "Designing with Flash Memory", Brian Dipert, Annabooks, 1993.

[70] "Non-volatile memory for fast, reliable file systems", Mary Baker, Satoshi Asami, Etienne Deprit, John Ouseterhout and Margo Seltzer, *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, October 1992, p.10.

[71] "A Flash-Memory Based File System", Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda, *Proceedings of the 1995 Usenix Technical Conference*, January 1995.

[72] "Cleaning policies in mobile computers using flash memory", M.-L. Chiang and R.-C. Chang, *Journal of Systems and Software*, **Vol.48**, **No.3** (1 November 1999), p.213.

[73] "High-Speed DRAM Architecture Development", Hiroaki Ikeda and Hidemori Inukai, IEEE Journal of Solid-state Circuits, Vol.34, No.5 (May 1999), p.685.

[74] "A Four-Level Storage 4-Gb DRAM", Takashi Okuda and Tatsunori Murotani, *IEEE Journal of Solid-state Circuits*, **Vol.32**, **No.11** (November 1997), p.1743.

StackGhost: Hardware Facilitated Stack Protection

Mike Frantzen
CERIAS
frantzen@cerias.purdue.edu

Mike Shuey
Engineering Computer Network
shuey@ecn.purdue.edu

Abstract

Conventional security exploits have relied on overwriting the saved return pointer on the stack to hijack the path of execution. Under Sun Microsystem's Sparc processor architecture, we were able to implement a kernel modification to transparently and automatically guard applications' return pointers.

Our implementation called StackGhost under OpenBSD 2.8 acts as a ghost in the machine. StackGhost advances exploit prevention in that it protects every application run on the system without their knowledge nor does it require their source or binary modification.

We will document several of the methods devised to preserve the sanctity of the system and will explore the performance ramifications of StackGhost.

1 Introduction

This paper presents a simple but elegant solution to the now infamous buffer overflow and some primitive format string attacks. Most security exploits have traditionally overwritten a function's saved return address. The attacker can then direct the flow of execution into an arbitrary instruction stream that is invoked when the vulnerable function tries to return control to its caller.

By taking advantages of one of the nuances of Sun Microsystem's Sparc processor architecture, we were able to engineer a kernel modification to OpenBSD 2.8 to help safeguard the return address. The kernel modification performs transparent, automatic and atomic operations on the return address before it is written to the stack and before the function transfers execution back to the saved return address.

Knowledge of what buffer overflows are [12], their relevance to security exploits [1, 13] and why they occur is a prerequisite to understanding this paper.

Section 2 describes the architectural issues involved in StackGhost. Section 3 details the implementation. Section 4 describes the performance effects. Section 5 acknowledges the limitations. Section 6 hypothesizes on extesion to other architectures. Section 7 describes the related research. Finally, Section 8 presents our conclusions.

2 Architectural Issues

2.1 Conventional Function Calls

Four bulk operations are performed to call a function in a conventional architecture. The function's parameters are saved onto the stack. The caller's registers are also saved onto the stack to prevent corruption by the callee. The instruction address is saved for the called function to return back to once it is finished. And only then can execution be transfered to the function.

Once the function completes its task, it jumps back to the return address saved on the stack.

2.2 Sparc Function Calls

2.2.1 Register Windows

When the Sparc architecture was first designed, the overhead associated with saving registers to the stack during a conventional function call was believed to be very large, or at least significant enough to warrant architectural changes to speed this process. Rather than wasting valuable CPU cycles to copy register data to and from the stack, Sparc architects attempted to provide hardware mechanisms to ensure that a function call gets a private set of registers for the duration of the function. When the function completes, the previous set of registers return to existence with (in most cases) no interaction with the stack whatsoever.

During normal execution, a Sparc processor has 32 visible general-purpose integer registers. These registers are divided into four groups based on the sort of data they are to contain, according to the Sparc Application Binary Interface (ABI) [17]:

global registers for data common across function calls.

input registers for incoming function parameters (including the frame pointer and return pointer).

local registers for general use.

output registers for parameters to deeper called functions, the return value from deeper function calls, the stack pointer, and the saved program counter after a jump and link.

The latter three groups (input, locals, and output) comprise a register window.

When a function is called, it allocates a new window for its specific use. The global registers are shared between both the old and the new windows (meaning that any modification of global data in the callee will be visible in the caller). The callee receives a new group of local registers, as well as a new set of output registers - these registers are not accessible from the calling function. Finally, the caller's output registers are rotated to be the input registers for the called function. Any changes the callee should make to its input registers will be visible to the caller as changes in the caller's output group of registers.

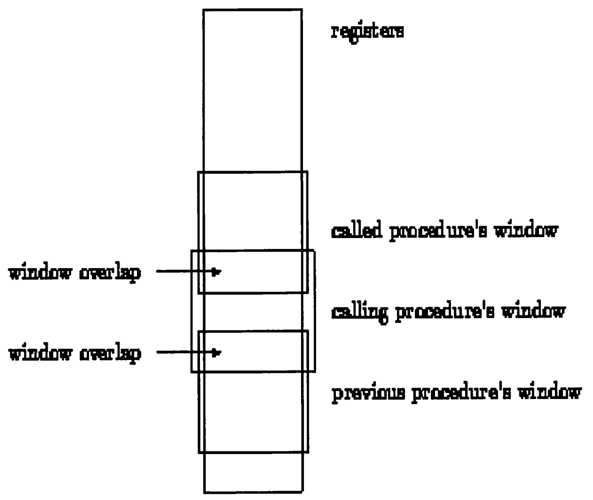

registers
called procedure's window
window overlap
calling procedure's window
window overlap
previous procedure's window

Figure 1: Register Window Overlap

In this way parameters can be passed from one function to another without (usually) interaction with the stack. The caller's code need only put parameters in its output registers, then call a function. The called function will have access to the caller's output registers in its own input registers. Return values are the reverse of this process; the called function leaves the return value in a particular input register, which then reverts to being an output register for the caller as soon as the function returns.

Nested function calls will create a chain of linked register windows. Each function call will use the same group of eight global registers, but will have its own group of eight local registers for its own private use. The output registers from the first function will be the input registers for the second deeper function called; the outputs from the second will be the inputs for the third, and so on.

Obviously, this trend can't go on forever. Each register window involves 24 registers (8 input, 8 local, 8 output), a third of which are shared with the calling function and two thirds of which need to be allocated by the processor. (The global registers are not shifted.) The processor will only have a limited number of registers available - most modern Sparc processors provide enough for seven or eight windows - and eventually some registers must be reclaimed.

The job of reclaiming registers falls to the operating system. When the number of allowable windows is about to be exceeded (as will occur with any program exhibiting deeply-nested or recursive functions) a register window overflow interrupt is generated. The OS will respond by copying the oldest register window onto the stack, relocate the now defunct register window, and return control of execution to the program without it knowing it missed a beat. Eventually the deeply-nested functions in the program will start to complete but the caller's registers will be defunct and need to be fetched. The processor will generate a register window underflow interrupt and force the OS to restore the previously saved registers.

This OS interaction provides the basic hardware primitives needed for StackGhost's operation. In a conventional function call architecture, there is no feasible way for the OS to automatically examine critical areas of the stack as they are being written. However, because the OS is ultimately in charge of when registers are written to the stack on the Sparc architecture, it is possible to take extreme precautions to ensure the security of critical data, such as the return address and frame pointer.

2.2.2 Memory Alignment

StackGhost also takes advantage of one other Sparc architectural feature. Instructions must be aligned on a four byte boundary. Otherwise, the hardware traps (interrupts) into the alignment fault handler in the kernel which kills the process. We will revisit this architectural requirement to enhance an attack detection algorithm.

3 Implementation

The beauty of StackGhost is that it doesn't have to operate on every function call. StackGhost only needs to be invoked in deep function call sequences or recursive programs – when the program overflows or underflows the register windows and thus interact with the stack. If a program only performs shallow function call sequences, StackGhost may never be invoked to write register windows to the stack.

The hardware ultimately has the responsibility to decide when a register window needs to be written to or read from the stack. When the decision occurs, the processor automatically invokes the overflow or underflow handlers and StackGhost in the process.

To maximize the security afforded by StackGhost, the mechanism either needs to prevent a corrupt return address from exploiting oversights in the program or it needs to detect corruption of the return pointer. Unfortunately, one does not guarantee the other.

In order for a corrupt return pointer to exploit the system, the return pointer must actually be pointed carefully back into exploit code. A reversible transform can be applied to the legitimate return address and the result written to the process stack. When the return pointer needs to be accessed, the reverse transform can be applied before the access completes. Thus the value saved on the stack is actually a computation of the real return address. To retrieve the real value, the inverse computation is calculated. If an attacker does not know the transform or the key to the transform, he or she cannot predictably affect execution.

There are two ways to transparently detect a corrupt return pointer. The first is a function of the above transform function. Since the Sparc processor requires that all instructions be aligned on a 32bit word boundary, the lower two bits of an instruction's address will both be zero. The transform can invert one or both of the two least significant bits. A corrupt return pointer on the stack will be detected if those bits are not set at the time of the inverse transform. The Sparc processor will take care of this detection in hardware and cause an alignment trap.

A corrupt return pointer can also be detected by keeping a return-address stack. Every time a return pointer is saved to the program stack, the handler can keep another copy in memory not accessible to the userland process (a return-address stack). A corrupt return pointer is detected if a function tries to return to a location not saved in the return stack.

3.1 Per-Kernel XOR Cookie

The most trivial incarnation of return pointer protection consists of XORing a fixed cookie into the return pointer. XORing the cookie with the pointer before it is saved onto the stack and again after it is popped off preserves the legitimate pointer but will distort any attack.

By setting one or both of the two least significant bits (LSBs) in the cookie, the XOR not only inhibits exploit, it also can detect a corrupt return pointer. If an attacker does not know which of the LSBs are set on the stack, the corrupt return pointer will cause an alignment fault unless the attacker gets lucky. Even if each state of the LSBs are tried, the interaction with the remainder of the cookie shound hinder the predictable operation of a corrupted return pointer.

A Per-Kernel XOR cookie can be built into OpenBSD by adding about a dozen assembler instructions. A sign-extended 13-bit cookie can be built into the kernel as an immediate operand that will cost one extra instruction per window pushed and another instruction when the window is popped.

The Per-Kernel XOR cookie can also be trivially defeated. It is constant for every process on the system. The cookie can be determined if an attacker is able to run arbitrary programs. Even without a priori knowledge of the cookie, an attacker could use a large shell code sled to slide into the main exploit code [1].

The Per-Kernel XOR cookie will not be enough to stop a competent attacker.

3.2 Per-Process XOR Cookie

A safer alternative to a per kernel XOR cookie would be to use a different random cookie for every process. The cookie can be stored in the Process Control Block (PCB) outside of user readable memory. The PCB will be automatically copied on a fork() and re-initialized on an execve(). There is even an extra 32-bit padding field in the OpenBSD PCB structure that can be used to store the 32-bit cookie.

A Per-Process XOR Cookie is far safer than per kernel granularity. But the XOR cookie can be inferred if an attacker can read the distorted return pointer off the stack and can also predict what the real return pointer should be. Format string vulnerabilities allow the first condition and looking at the vendor supplied application binary can often provide the real return pointer.

The Per-Process Cookie overhead will add approximately four instructions of overhead to both the push and pop action. In a few instances it will be as few two when the PCB pointer is already available in a register.

3.3 Encrypted Stack Frame

We can further mitigate detection of a corrupt return pointer with a more unpredictable transformation of the return pointer. We have the option of encrypting part of the stack frame when the window is written to the stack and decrypting it during retrieval.

Unfortunately there are several major obstacles to encrypting the return pointer.

1. Encrypting and decrypting every frame may seriously hinder performance.
2. At best, there are only 16 registers to work with. Auxiliary space would have to be statically allocated in the PCB.
3. The algorithm cannot rely on block chaining since userland threading or setjmp-longjmp could shuffle the call-return ordering.
4. The plaintext is easily predictable. Most of the high bits of the frame pointer will be set. Most of the high bits of the return pointer will be zero. The input registers (function arguments) will be fairly constant.

We believe a 64-bit block algorithm would offer improved security over the XOR cookie methods described above by using the concatenation of the frame pointer and return pointer as the input to the encryption algorithm. It could be a cryptographically **weak** usage but would stop all but the most determined adversaries. Encrypting the stack frames would unfortunately impose significant performance degradation for obvious reasons.

The encryption algorithm would have to be modified to encrypt the stack frame if StackGhost must detect a corrupted return pointer. The previous two StackGhost methods used the two LSBs as a form of an in-band secret. Using encryption as the transform would obviously cause the LSBs to be random.

3.4 Return-Address Stack

The pinnacle incarnation of StackGhost would implement what processor architects call a "return-address stack". To improve unconditional branch prediction, modern processors keep a FIFO stack in silicon of the return addresses of function calls [15, 11, 3]. Every time a *CALL* instruction is executed, its return address is pushed onto the stack. Every time a *RETURN* instruction enters the pipeline, the next address is popped off the stack and the processor continues fetching from the associated address seamlessly. A few cycles later, the real return address will be established and the processor can recover from a misprediction if need be.

We shall describe the theory developed to date. Our design criteria are as follows:

1. The mechanism must break no standard or common software.
2. The mechanism must guarantee the detection of a smashed stack.
3. The mechanism must kill any process with a corrupt stack.
4. The mechanism must have negligible memory utilization.
5. The mechanism must be implementable and debugable.

An obvious first approach might be to build in a return-address stack as a FIFO queue just as is done in hardware. Unfortunately, something so simple would break userland threading, setjmp() and longjmp(), and possibly C++ exceptions. Setjmp(), longjmp() and C++ exceptions introduce the problem that multiple return addresses can be bypassed by a deep multi-level return. This can be solved by scanning the entire stack until the return address can be located. Userland threading introduces a similar situation. When a thread relinquishes the processor to a sibling thread, it switches to a seperate stack. The second thread may be at the apex of a deep calling sequence and start returning. Again the queue will be out of order instead of FIFO and may have to be walked for every return. Performance will be sacrificed. If a thread is terminated or a program longjmp()'s, queue entries will reference stale stack frames and persist until the process terminates.

A more refined approach to designing a return-address stack is to add a small hash table in the PCB. Every time a register window needs to be cleansed, the mechanism would add an entry into the hash table (indexed off the base address of the stack frame). And then store the base address to use as the comparison tag, the return pointer, and a random 32-bit number. In the place of the return address in the stack frame, it would place a copy of the random number. When StackGhost retrieves the stack frame to refill the register window, it can compare the random number on the stack with its image in the hash table. If the instances do not match, an exploit has occurred and the program must be aborted. Otherwise, StackGhost fills the register return address with the one stored in the hash table.

A return-address hash table alleviates the performance problems associated with userland threading but does not address the memory leak associated with setjmp and longjmp or a terminated thread. Fortunately, setjmp and longjmp are both assisted by the kernel as a system call. Upon receiving the longjmp syscall, the kernel can walk backwards through the stack until the setjmp location is found, removing the hash entries along the way. An indirect benefit of walking the stack is that it also helps secure the jmpbuf (setjmp storage buffer).

For operating systems other than OpenBSD that support symmetric multiprocessing on Sparc and with kernel managed threads, mutual exclusion would have to be guaranteed at some level on the hash table. A locking primitive per window overflow and underflow handler invocation may prove prohibitivly expensive.

Further testing in a careful multi-user environment would be needed.

4 Performance Effects

4.1 Micro Benchmarks

Micro benchmarks were run under each of StackGhost's protection mechanisms and the results appear above in Figure 2 (see appendices for benchmark code and details). For the Return-Address stack mechanism, an optimistic approximation was implemented. It assumed an adequate number of pre-allocated entries its the free list and a naive

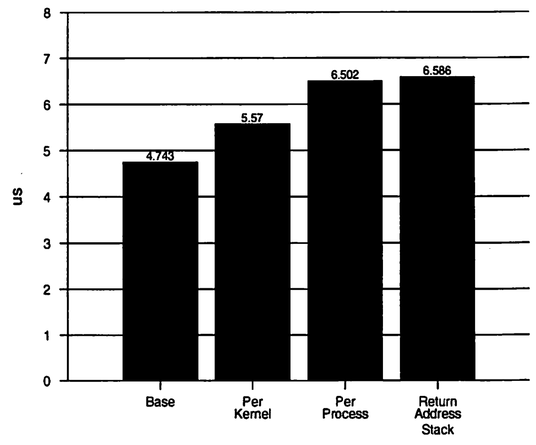


Figure 2: Microseconds per Function Call

random number generation scheme. Both Cookie methods are the true StackGhost implementations.

The micro benchmarks show a worst case scenario with a deeply recursive instance of an eight instruction function. Each of the function calls will invoke StackGhost. On a 70Mhz Sparc 4, the Per-Kernel XOR cookie imposes a little under one microsecond per function call penalty. The Per-Process cookie StackGhost overhead is a little under two microseconds per call. The return-address stack cost negligably more than the Per-Process mechanism.

In the **absolute** worst case (shortest possible recursive function that will still return), the Per-Kernel XOR cookie causes a 17.44% overhead over the baseline. The Per-Process XOR cookie can result in a 37.09% overhead. The return-address stack approximation imposes a 38.86% overhead.

Again, it **cannot** get worse unless there are unwieldy cache or TLB affects. We speculate that a bulk of the overhead is actually attributable to an additional TLB and cache miss instead of the additional instruction count.

The performance penalties could be reduced if the StackGhost code was interleaved into the trap handlers instead of just inserted. Sparc processors are superscalar, albeit in-order, and can take advantage of some instruction level parallelism (ILP). If the trap handlers themselves were re-written to increase ILP, the optimization should absorb most of the StackGhost cost.

4.2 Macro Benchmarks

The SPEC95 integer benchmark suite was also run to establish macro benchmarks (see Appendix 1 for environmental details). The results appear below in Figure 3.

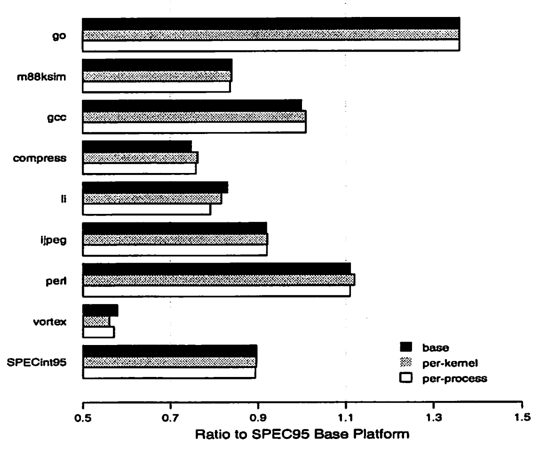


Figure 3: SPECint95 benchmarks

The performance penalties measured by the SPEC95 integer benchmark suite indicated that StackGhost only shaved a few hundreths of a point off the speed metrics. The performance aberrations may be more attributable to noise than the StackGhost mechanism. An un-StackGhosted OpenBSD 2.8 kernel had slightly worse performance than a StackGhosted version in some instances, the only other explanation is that it is due to cache or TLB effects.

Discounting any noise in the benchmark, the geometric mean SPEC rating showed a StackGhost overhead of 0.1% with a Per-Kernel cookie and a 0.4% overhead with a random Per-Process cookie.

5 Limitations

There are several moderate to serious deficiencies in StackGhost. Some could be diminished with further research but others are inherent.

5.1 Unpredictable Execution

The StackGhost XOR Cookie methods of hindering exploits do not always detect the corruption of the stack. If the attacker's return pointer manages to align correctly after being XORed with the cookie, execution will be transfered in an unpredictable manner.

Execution may divert to a random but legal stretch of code and cause data corruption. Of course, a successful attack may have the same chance of causing data corruption since no cleanup code will be called anyway.

5.2 Forked Processes

In the current StackGhost incarnation, forked processes have an identical Per-Process cookie.

It may be possible to unroll the stack and adjust each return pointer in the new process. But the process would have to be non-threaded and it would duplicate the program stacks instead of using a copy-on-write mechanism – potentially driving up memory utilization. Again, further research must be done.

5.3 Rootshell vs. DoS

If StackGhost saves a network daemon from a successful attack, it should abort the network daemon. A rootshell exploit will just be converted into a denial of service exploit since the daemon will be down. This behavior is an added incentive to remedy the underlying problems, instead of just mitigating exploit.

5.4 Random Pool Depletion

If random Per-Process keys are used, bursts of rapid program spawning could deplete the randomness pool. A starved pool could hinder other programs from executing until more randomness can be gathered.

5.5 Debuggers

Userland debuggers are currently broken by the XOR cookies. They will not be able to backtrace since the in-core return pointers are obviously distorted. The in-kernel core dump mechanism may be able to walk the stack and cleanse each activation record in the program. Further research must be done. Threaded programs would present an additional beast for reasons outlined above by the kernel return stack.

Debugging via Ptrace() will also present problems for the parent processes since the in-core program counter will have been modified by StackGhost.

5.6 Granularity

The current implementation of StackGhost protects each userland process on the system. It may be desireable to selectively protect processes deemed to be "at risk." Setuid, setgid and otherwise privileged processe are the likely candidates for automatic coverage. The XOR cookie mechanisms of StackGhost may disable coverage by using a NULL cookie since XORing any number with zero is the equivalent of adding by zero – no effect.

5.7 Unaffected Exploits

StackGhost will not stop every exploit, nor will it guarantee security. Exploits that StackGhost will **not** stop include:

1. A corrupted function pointer (atexit table, .dtors, etc.)
2. Data corruption leading to further insecure conditions.
3. "Somehow" overwriting a frame pointer with a frame pointer from far shallower in the calling sequence. It will short circuit backwards through a functions' callers and may skip vital security checks.

6 Other Architectures

The application of the StackGhost mechanisms to architectures other than Sparc is contingent on a trap into the kernel when registers are pushed onto the stack and again when they are popped off. We know of no common architectures which provide the convenience of the Sparc register window overflow and underflow traps. But we believe there are several architectural features which could approximate the behavior of Sparc.

6.1 Hardware Breakpoints

Many architectures provide hardware assisted debugging in some fashion. If the support comes in the form of hardware breakpoints, and the breakpoints can be placed on memory accesses (aka watchpoints) instead of instruction addresses, the kernel may be able to load a breakpoint on the address where the next and last return pointers are stored before context switching into a processes. Access to the current function's return pointer and the next function's in the calling sequence would both cause a trap into the kernel. A deeper function call will cause a trap which must add a breakpoint to the next return pointer location when it saves the current. A return will cause a trap which must confirm that there is a breakpoint on the next previous return pointer.

The use of hardware breakpoints to approximate the trap behavior of register windows requires knowledge of stack layout a priori or the userland process must include stack layout hints.

6.2 Page Protection

Most architectures allow some protection mechanism to limit access to virtual pages. By marking stack pages as unaccessible (for both reads and writes), the kernel could guarantee a trap every time the stack is accessed. Unfortunately, the kernel will be notified for every stack access. A variation of this method proved too costly on IA32 under the MemGuard implementation [4] briefly described later.

6.3 IA64

Intel's IA64 architecture supports variably sized register windows (the *Register Stack Engine* in the Intel vernacular) [7, 8, 9, 10]. In IA64, function's can request an arbitrary number of registers unlike the 24 register window on Sparc. If an overflow or an underflow occurs, the processor stalls while the hardware interacts with the backing store. The actual loading or storing of the registers is done by the hardware instead of by kernel trap handlers.

To simulate the Sparc register window trap behavior, it may be possible to misalign the backing store pointer. Every time the Register State Engine stores to the backing store or retreives registers from it, there will be a trap into the kernel thus invoking the StackGhost mechanism.

7 Related Work

There have been several prior research endevours against buffer overflows and to guard the stack. This is by no means an exhaustive list.

7.1 StackGuard

Crispin Cowan's StackGuard is a modified compiler which places canaries (the term canary can be used interchangeable with our use of the term cookie) around the return pointer in function prolog. A buffer overflow will modify the canary on its way to overwriting the adjacent return pointer. If the function epilog detects a dirty canary, it rightly infers that an exploit has occurred, it logs the exploit and it aborts the program [4].

StackGuard can also XOR a random canary into the return address in the function prolog and XOR the canary out in the epilog. This should cause an undetected corrupt return pointer to dump core instead of executing the exploit code.

Another technique called MemGuard was described in the same paper as StackGuard. MemGuard designates the return address on the stack of an x86 machine as a "quasi-invariant." It only allows a store to that memory location through the MemGuard API. This involved marking the entire stack page read-only during function prolog, and unprotecting the page during the epilog. A special trap handler was installed in the kernel to emulate the writes to the stack locations near the return address that were unfortunate enough to fall on the same virtual memory page. MemGuard proved to impose an inordinate overhead.

7.2 StackShield

StackShield works as an assembler processor supported by the GNU C and C++ compilers. It works by modifying the function prolog to store away the return pointer into a stack distant enough that overflow is not likely. Upon function return, the function epilog actually returns from the location specified in the private return stack instead of the program stack [18]. The only exploit detection StackShield performs is checking the segments function pointers point to.

7.3 ProPolice

Hiroaki Etoh's ProPolice is a modification to the GNU C compiler that places a random canary between any stack allocated character buffers and the return pointer [5]. It then validates that the canary has not been dirtied by an overflowed buffer before the function returns. ProPolice can also reorder local variables to protect local pointers from being overwritten in a buffer overflow.

7.4 LibSafe

LibSafe is a library modification to Linux that safely wraps functions known to be "unsafe" and contains any damage to the local stack frame [2]. Also included in the LibSafe paper is a tool called LibVerify that will rewrite a binary application to perform a return address check.

7.5 Non-Exec pages

There are several implementations available that attempt to hinder an exploit by limiting the memory segments that code can execute in.

Solar Designer architected a kernel modification to x86 Linux to prevent execution in stack pages. Exploits will not be able to run their own code if the buffer resides on the stack (which most buffer overflows do) [16]. Sun also built an optionally enabled non-executable stack into the Sparc version of Solaris.

PaX is a x86 Linux kernel modification to mark all data pages non-executable, not just stack pages. PaX inhibits heap exploits in addition to stack overflows [14].

There are several overflow exploits that non-executable pages do not inhibit. By far the most common is the "return into libc." Instead of executing custom exploit code, the attack directs the return pointer back into code can have malicious consequences depending on its arguements. The easiest example is to point the return address back at the system() library call and point the arguement at an instance of "/bin/sh".

8 Conclusion

StackGhost has proven to be an effective defense against common exploit techniques at a negligible cost to the user. StackGhost's primary merit is that it is a kernel modification and does not require mass recompilation or the administrative headaches of selective protection. The current implementation of StackGhost is deficient in that it cannot guarantee the explicit detection of a stack exploit, it can only foil the operation of an exploit.

When the seperate return stack apparatus of StackGhost is fully implemented, StackGhost will offer guaranteed detection of the traditional buffer overflow at a fraction of the cost of the other available stack protection mechanisms.

9 Availability

The StackGhost project homepage can be found at http://stackghost.cerias.purdue.edu. The relevent patches to OpenBSD shall be placed in the Public Domain.

10 Acknowledgments

We would like to thank Rick Kennell and Florian Kerschbaum for technical advice. And to thank Susan Hazel for assistance with paper organization and editing.

References

[1] Aleph One. "Smashing The Stack For Fun And Profit." *Phrack*, 7(49), November 1996.

[2] Arash Baratloo, Timothy Tsai, and Navjot Singh. "Transparent Run-Time Defense Against Stack Smashing Attacks." *Proceedings of the USENIX Annual Technical Conference, June 2000.*

[3] C. F. Webb. "Subroutine call/return stack." *IBM Technical Disclosure Bulletin, 30(11)*, Apr. 1988.

[4] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle and Qian Zhang. "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," *Proceedings of the 7th USENIX Security Conference*, January 1998, San Antonio, TX.

[5] Hiroaki Etoh.
"GCC extension for protecting applications from stack-smashing attacks."
http://www.trl.ibm.co.jp/projects/security/ssp

[6] Immunix.org. "StackGuard Mechanism: Emsi's Vulnerability."
http://immunix.org/StackGuard/emsi_vuln.html.

[7] Intel IA64 Architecture Software Developer's Manual. "Volume 1: IA-64 Application Architecture Revision 1.1." July 2000

[8] Intel IA64 Architecture Software Developer's Manual. "Volume 2: IA-64 System Architecture Revision 1.1." July 2000

[9] Intel IA64 Architecture Software Developer's Manual. "Volume 3: IA-64 Instruction Set Reference Revision 1.1." July 2000

[10] Intel IA64 Architecture Software Developer's Manual. "Volume 4: IA-64 Itanium processor Programmar's Guide Revision 1.1." July 2000

[11] Kevin Skadron, Pritpal S. Ahuja, Margaret Martonosi, Douglas W. Clark. "Improving Prediction for Procedure Returns with Return-Address-Stack Repair Mechanisms." *Proceedings of the 31st ACM/IEEE International Symposium on Microarchitecture,* Nov 1998, Dallas TX.

[12] Mudge. "How to Write Buffer Overflows."
http://l0pht.com/advisories/bufero.html, 1997.

[13] Nathan P. Smith. "Stack Smashing vulnerabilities in the UNIX Operating System."
http://millcomm.com/~nate/machines/security/stack-smashing/nate-buffer.ps, 1997.

[14] PaX Team. "NonExecutable Data Pages."
http://pageexec.virtualave.net/pageexec.txt

[15] S. McMahan. Cyrix Corp. "Branch Processing unit with a return stack including repair using pointers from different pipe stages." U.S. Patent No. 5,706,491. Jan, 1998.

[16] Solar Designer. "NonExecutable User Stack."
http://www.false.com/security/linux-stack/.

[17] SPARC International, Inc. "The SPARC Architecture Manual." Version 8. 1992.

[18] Vendicator. StackShield: A "stack smashing" technique protection tool for linux.
http://www.angelfire.com/sk/stackshield/.

Appendix 1: Benchmark Procedure

A short C program was used to microbenchmark the function call overhead imposed by StackGhost. It was compiled with gcc version 2.95.3 19991030 (prerelease).

The Spec95 integer suite was run to generate the macro benchmarks. The benchmark suite was built with gcc version 2.95.3 19991030 (prerelease).

All the benchmarks were run on a 70Mhz SparcStation 4, 32MB of ram, PROM Rev 2.20, and no L2 cache. The machine was operating in multi-user mode under fairly constant conditions for each iteration of the benchmarks.

Appendix 2: Micro Benchmark

```
#define DEPTH 10000
#define TRIALS 1000

void deep(int n)
{
  if (--n)
    deep(n);
}

int main(void)
{
  struct timeval start, stop;
  float total, times[TRIALS];
  int i;

  /* Prefault the stack */
  deep(DEPTH);

  for (i = 0; i < TRIALS; i++) {
    usleep(1); /* Give up time slice to avoid context switch */
    gettimeofday(&start, NULL);
    deep(DEPTH);
    gettimeofday(&stop, NULL);

    times[i] = stop.tv_sec - start.tv_sec + (float)(stop.tv_usec - start.tv_usec) / (1000000);
  }

  for (i = 0, total = 0; i < TRIALS; i++)
    total += times[i];

  printf("Avg time %.5fs\n", total / (float)TRIALS);
  printf("Avg us/call %.3fus\n", (1000000 * total) / (float)(TRIALS * DEPTH));
}
```

Improving DES Coprocessor Throughput for Short Operations

Mark Lindemann
IBM T.J. Watson Research Center
Yorktown Heights, NY 10598-0704
mjl@us.ibm.com

Sean W. Smith*
Dept. of CS/Institute for Security and Technology Studies
Dartmouth College
Hanover, NH 03755
sws@cs.dartmouth.edu, http://www.cs.dartmouth.edu/~sws/

Abstract

Over the last several years, our research team built a commercially-offered secure coprocessor that, besides other features, offers high-speed DES: over 20 megabytes/second. However, it obtains these speeds only on operations with large data lengths. For DES operations on short data (e.g., 8-80 bytes), our commercial offering was benchmarked at less than 2 kilobytes/second. The programmability of our device enabled us to investigate this issue, identify and address a series of bottlenecks that were not initially apparent, and ultimately bring our short-DES performance close to 3 megabytes/second. This paper reports the results of this real-world systems exercise in hardware cryptographic acceleration—and demonstrates the importance of, when designing specialty hardware, not overlooking the *software* aspects governing how a device can be used.

1 Introduction

What *is* "fast DES?" The challenge of *meaningfully* quantifying cryptographic performance has been a long-standing issue.

Over the past several years, our team has worked on producing, as a commercial offering, a cryptographic embedded system: a high-performance, programmable secure coprocessor platform [9], which could take on different personalities depending on the application program installed. This device featured hardware crypto support for modular math and DES in the original version, with outer-CBC TDES and SHA-1 added in the Model 2. Our initial commercial target was an application program [1] that turned the platform into a secure cryptographic accelerator.

Besides the physical and logical security of the device, our team prided itself on the fast DES (and, in the Model 2, outer-CBC TDES) that our device provided. Measured from an application program on the host (in order to give a more accurate figure), our initial device performed DES at about 20 megabytes/second; the follow-on does outer-CBC TDES at close to this rate. We note, however, that we were focused on *secure coprocessing*, and wanted fast DES in contexts where the keys and decisions were under the control of the trusted third party inside the box, not the less secure host. Two potential examples of such scenarios include re-encryption of a hardware-protected Kerberos database [3], and information servers that ensure privacy even against root [8].

However, these figures were for *bulk* performance: operations consisting of CBC encryption or decryption of input data that is itself megabytes long. For operations on short data, our device was *several orders of magnitude slower.* When an external colleague—who required large numbers of DES operations on inputs each 8-80 bytes—benchmarked our commercial offering, he only measured about 1.5 kilobytes/second. [5]

The programmability of our device enabled us to investigate this issue, and we assumed that our intimate knowledge of the internals would enable us to immedi-

*This work was supported in part by the U.S. Department of Justice, contract 2000-DT-CX-K001.

ately identify and rectify the bottleneck. This assumption turned out to be incorrect. In this paper, we report the lengthy sequence of experiments that followed. We finally improved short-DES performance by three orders of magnitude over the initial benchmark, but have been continually surprised at where the bottlenecks really were.

We offer this contribution as a real-world systems exercise in cryptographic acceleration. It demonstrates the value of programmability in a cryptographic accelerator—because without this flexibility, we would not have achieved the three orders of magnitude speedup. More importantly, it demonstrates the importance of considering *how a system will actually be used*, and *how the control data will be routed*, when designing specialty cryptographic hardware. Far too often, the hardware design process leaves these issues for post-facto software experimenters (like ourselves) to discover. Consequently, our work also offers some potential lessons for future design of hardware intended to accelerate high-latency operations on small data lengths, as well as for the future design process.

2 System Background

Our device is a multi-chip embedded module, packaged in a PCI card. In addition to cryptographic hardware, and circuitry for tamper detection and response, we have a *general-purpose computing environment*: a 486-class CPU, executing software loaded from internal ROM and FLASH. Two generations of the device exist commercially; the older Model 1 and the newer Model 2. We did our experiments on the Model 2 (since that is all we had); discussions of principles that apply to both models do not specify a model number.

2.1 Software

The multiple-layer software architecture consists of foundational security control (Layer 0 and Layer 1), supervisor-level system software (Layer 2), and user-level application software (Layer 3). (See Figure 1.)

Our Layer 2 component [2] was designed to support application development. Within Layer 2, a kernel provides standard OS abstractions of multiple tasks and multiple address spaces; these abstractions support independent *managers:* components within Layer 2 which

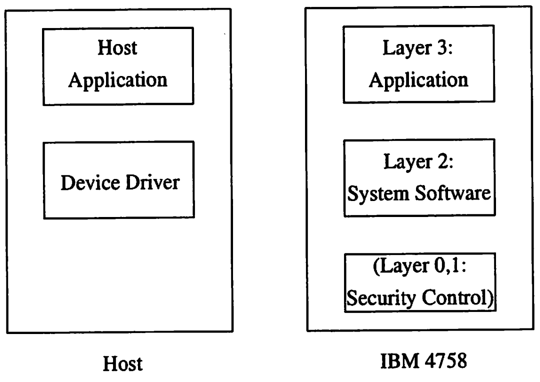


Figure 1 The software architecture of the coprocessor. The host software on the left runs on the host system; the card software on the right runs on the 486 inside the coprocessor.

handle cryptographic hardware and other I/O on the bottom, and provide higher-level APIs to the Layer 3 application on top.

Typically, this Layer 3 application provides the abstraction of its own API to host-side application. Figure 2 through Figure 4 shows the interaction of software components during applications such as standard DES acceleration:

(Figure 2) When it wants to use a service provided by the card-side application, the host-side application issues a call to the host-side device driver. The device driver then opens an `sccRequest` to the Layer 2 system software on the device. Layer 2 then informs the Layer 3 application resident on the device of the existence of this request, and some of the parameters the host sent along with it.

(Figure 3) The Layer 3 application then handles the host application's request for service; in this example, it directs Layer 2 to transfer data and perform the necessary crypto operations.

(Figure 4) The Layer 3 application then directs Layer 2 to close out the `sccRequest` and send the results back to the host.

2.2 Hardware

One of the many goals of our device was fast cryptography. As part of this goal, we included a FIFO/state machine structure that can transport data quickly into and out of an algorithm engine. Figure 5 shows how this pro-

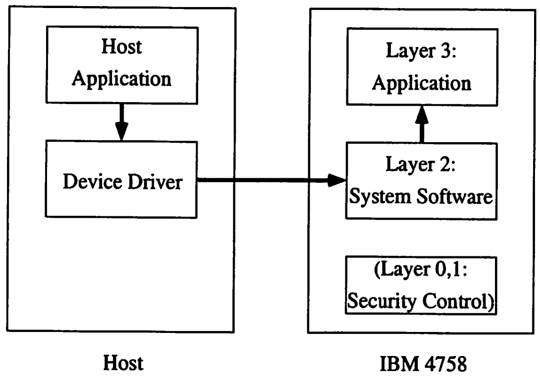


Figure 2 The host application opens an `sccRequest` to the application layer in the card.

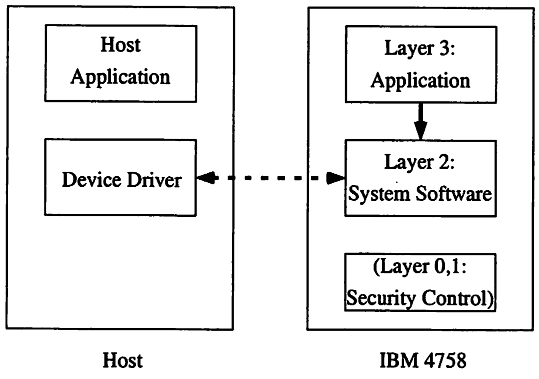


Figure 3 For standard external-external DES, the application layer asks Layer 2 to perform the operation; Layer 2 then directs the the data transfer.

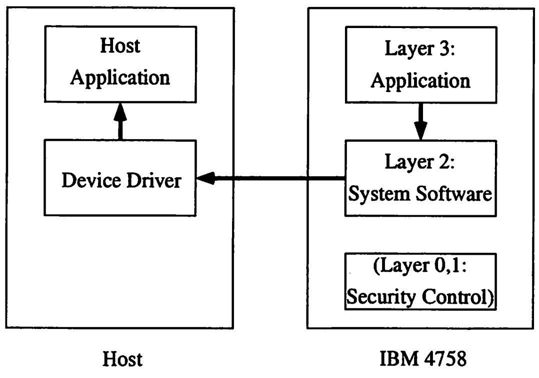


Figure 4 The application layer closes out the `sccRequest`, and sends the output back to the host application.

prietary FIFO structure works with the DES/TDES engine. (In our Model 2 hardware, this FIFO structure also supports fast SHA-1; in principle, this structure could be applied to any algorithm engine.)

For both input and output, we have two pairs of FIFOs—a *PCI FIFO* and an *internal FIFO*, for fast external and internal data transfer, respectively. We also have a *DMA controller*, for CPU-free transfer into and out of internal DRAM. These components enable the device CPU to arrange to do fast data transfer through the various on-board devices, without the active involvement of the CPU after the initial configuration. For example, to support fast bulk DES when the source and destination are both outside the device, the internal CPU can configure these components to support an external-to-external data path (PCI Input FIFO to Internal Input FIFO to DES, then back through the output FIFOs), load the relevant operational parameters (e.g., key, IV, mode) into the DES engine, and then let the the hardware move data through on its own.

Besides external-to-external DES, other common configuration paths include internal-to-internal bulk DES (Output DMA to Internal Input FIFO to DES, then back), and DMA transfer (e.g., PCI Input FIFO to Internal Input FIFO to Input DMA and vice versa). (Additionally, the DES hardware can be configured in bypass mode, but the commercial Layer 2 software does not use it.)

As an artifact of the hardware design, we have one principal constraint: *both* internal FIFO-DES paths must be selected (*bulk mode*), or *neither* must be selected (*non-bulk mode*).

However, changing between these modes resets the Internal FIFOs, and during non-bulk mode, the CPU has no way to restrain the Internal Input FIFO from filling to capacity.

Examples Figure 6 through Figure 10 show some examples of how the FIFO hardware supports card applications.

- (Figure 6) When the host application opens up an `sccRequest` to the card application, the card typically brings the input data into a DRAM buffer via DMA.

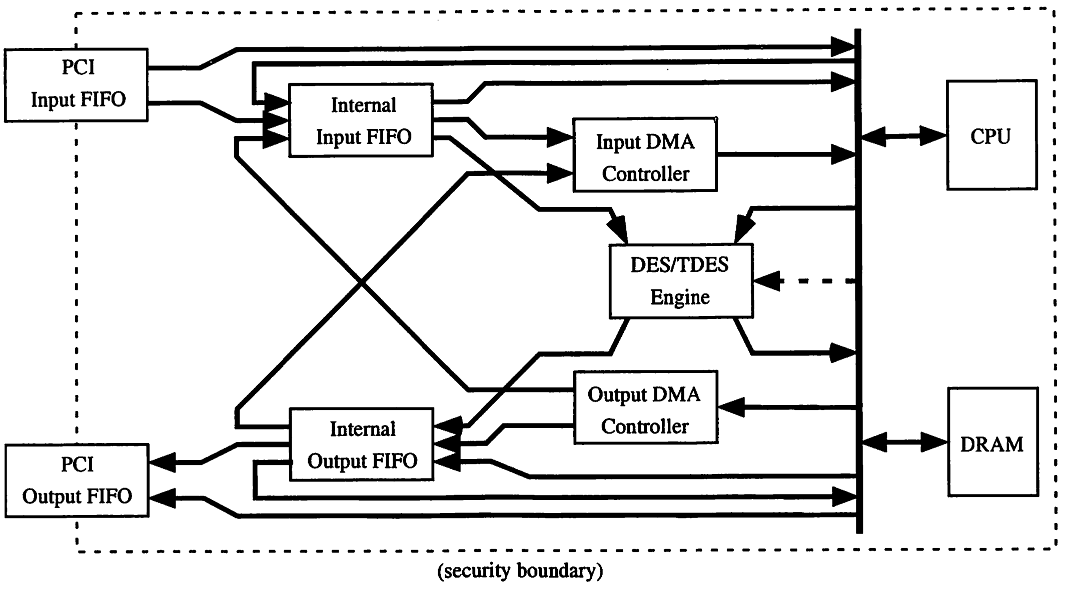


Figure 5 The FIFO structure supporting DES/TDES, within the coprocessor.

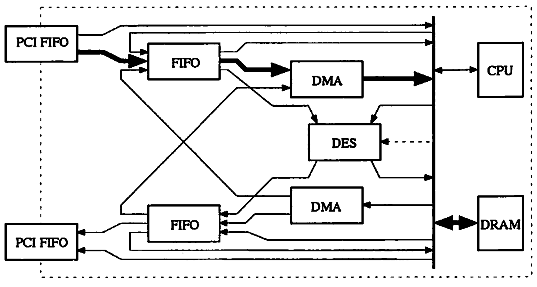


Figure 6 The bold arrows show how the internal CPU can configure the FIFOs to bring data into the card via DMA.

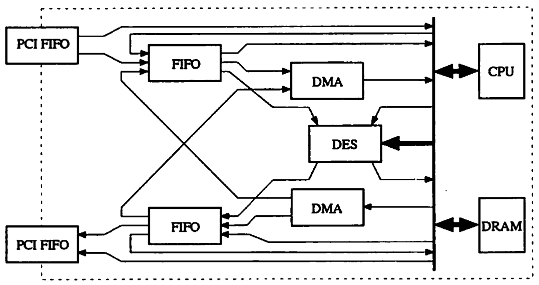


Figure 7 The bold arrows show how the internal CPU can load operational parameters into the DES chip from DRAM.

- (Figure 7) For a DES request, the card may then transfer the operational parameters from DRAM into the DES chip.

- (Figure 8) If the DES request is for external-external DES, the card will then configure the FIFOs to bring the data in from the host, through the DES chip (operating with the parameters we just loaded), then back to the host.

- (Figure 9) If the DES request is for internal-internal DES (but is too short to justify DMA), the card may just manually push the bytes through.

- (Figure 10) When the `sccRequest` is complete, the card may send the results back out to the host via DMA.

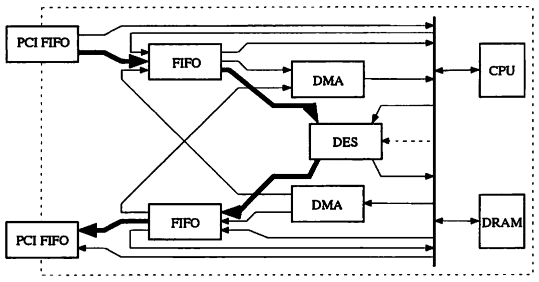


Figure 8 The bold arrows show how the internal CPU can configure the FIFOs to stream data from the host, through the DES chip, then back out to the host.

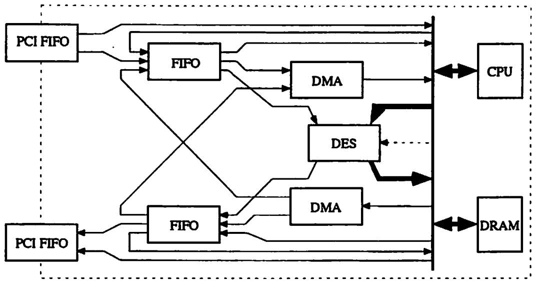


Figure 9 The bold arrows show how the internal CPU can also drive data from DRAM through DES via programmed I/O.

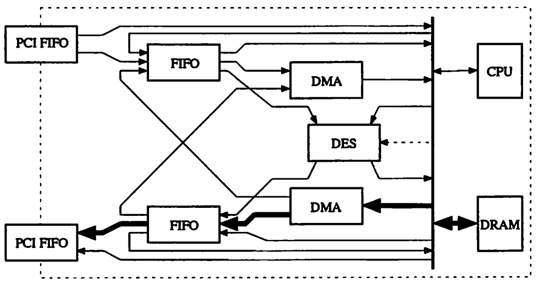


Figure 10 The bold arrows show how the internal CPU can configure the FIFOs to send data from DRAM back the host via DMA.

3 The Experiment Sequence

This unfunded "skunkworks" project had several goals: to try to see why the huge gap existed between what a colleague (using slower Model 1 hardware) measured for short-DES and what we measured for longer bulk DES; to try to improve the performance, if possible; and to explore migration of these changes (if the performance improves significantly) back into our commercial Layer 2 software (e.g., via some new "short-DES" API it provides to Layer 3).

But as a side-effect, we had a constraint: due to funding limitations (that is, zero funding) and the long-term goal of product improvement, we had to minimize the number of components we modified. For example, modifying the host device driver, even just to enable accurate latency measurements, was not feasible; and any solution we considered needed to be a small enough delta that a reasonable chance existed of moving it into the real product.

Since the colleague's database application (as well as the general nature of the problems to which we apply our secure coprocessing technology) required no exposure of key material, we did not measure host-only DES.

3.1 The Gauntlet is Thrown

Our colleague prompted this work when he demonstrated just how poorly our device performed for his application. Thus, to start our investigation, we needed to nail down the nature of the "DES" performance that he benchmarked at approximately 1.5 kilobytes/second.

This figure was measured from the host-side application program (recall Figure 1), using commercial Model 1 hardware with the IBM *Common Cryptographic Architecture (CCA)* application in Layer 3. (CCA also inserts a middle layer between the host application and the host device driver).

The DES operations were CBC-encrypt and CBC-decrypt, with data sizes distributed uniformly at random between 8 and 80 bytes. The IVs and keys changed with each operation; the keys were TDES-encrypted with a master key stored inside the device. Encrypted keys, IVs, and other operational parameters were sent in with each operation, but were not counted as part of the data throughput. Although the keys may change with each operation, the total number of keys (in our colleague's application, and in others we surveyed) was still fairly small, relative to the number of requests.

Experiment 1: Establishing a Baseline

Idea. We first needed to establish a baseline implementation that reproduced our colleague's set-up, but in a setting that we could instrument and modify. Our colleague used commercial Model 1 hardware and CCA; in our lab, we had neither, but we did have Model 2 prototypes. So, we did our best to simulate our colleague's configuration.

Experiment. We built a host application that generated sequences of short-DES requests (cipherkey, IV, data); we built a card-side application that: caught each request; unpacked the key; sent the data, key, and IV down to the DES engine; then sent the results back to the host. Figure 11 shows this operation.

Results. With this faster hardware (and lighter-weight software) than our colleague's set-up, we measured 9-12 kilobytes/second (with the speed decreasing, oddly, as the number of operations increased).

We chose keys randomly over a small set of cipherkeys. However, caching keys inside the card (to reduce the extra TDES key-decryption step) did not make a significant performance improvement in this test.

Experiment 2: Reducing Host-Card Interaction

Idea. Within our group, well-established folklore taught that each host-card interaction took a huge amount of time. Consequently, we first hypothesized that the reason short DES was so much slower than longer DES was because of the much greater number of host-card interactions (one set per each 44 bytes of data, on average) that our short-DES implementation required.

Experiment. We re-wrote the host-side application to batch a large sequence of short-DES requests into one `sccRequest`, and then re-wrote the card-side application to: receive this sequence in one step; process each request; and send the concatenated output back to the host in one step. Figure 12 shows this operation.

Results. We tried a several data formats here. Speeds ranged from 18 to 23 kilobytes/second (and now up to 40 kilobytes/second with key caching). This approach was an improvement, but still far below the apparent potential—host-card interaction was not the killer bottleneck.

Experiment 3: Batching into One Chip Operation

Idea. Another piece of well-established folklore taught that resetting the DES chip (to begin an operation) was expensive, but the operation itself was cheap. Until now, we had been resetting the chip for each operation (again, once per 44 bytes, on average).

Our next step was to see how fast things would go if we eliminated these resets.

Experiment. For purposes of this experiment, we generated a sequence of short-DES operation requests that all used one key, one direction ("decrypt" or "encrypt"), and IVs of zero (although the IVs could have been arbitrary). Our card-side application now received the operation sequence and sent it all down to the Layer 2 software. In Layer 2, we rewrote the DES Manager (the component controlling the DES hardware) to set up the chip with the key and an IV of zero, and to start pumping the data through the chip. However, at the end of each operation, our modified Manager did the proper XOR to break the chaining. (E.g., for encryption, the software manually XOR'd the last block of ciphertext from the previous operation with the first block of plaintext for the next operation, in order to cancel out the XOR that the chip would do.)

Results. Much to our surprise, we now measured as high as 360 kilobytes/second. Was DES-chip reset the killer bottleneck?

Distrusting folklore, we modified the experiment to reset the DES chip forh each operation anyway, and the top-end speed dropped slightly, to 320 kilobytes/second. So, it wasn't the elimination of chip resets that was saving time here.

Experiment 4: Batching into Multiple Chip Operations

Idea. How many Layer 3-Layer-2 context switches are necessary to handle the host's batched operation request?

Besides reducing the number of chip resets, the one-reset experiment of Experiment 3 also reduced the context switches from $O(n)$ to $O(1)$ (where n is the number of operations in the batch). The good performance of the multi-reset variant suggested that perhaps these context switches were a significant bottleneck.

Experiment. We went back to the multi-key, non-zero-IV set-up of Experiment 2, except now the card-side application sends the batched requests down to a modified DES manager, which then processes each one (with a chip reset and new key and IV each time). Figure 13 shows this operation.

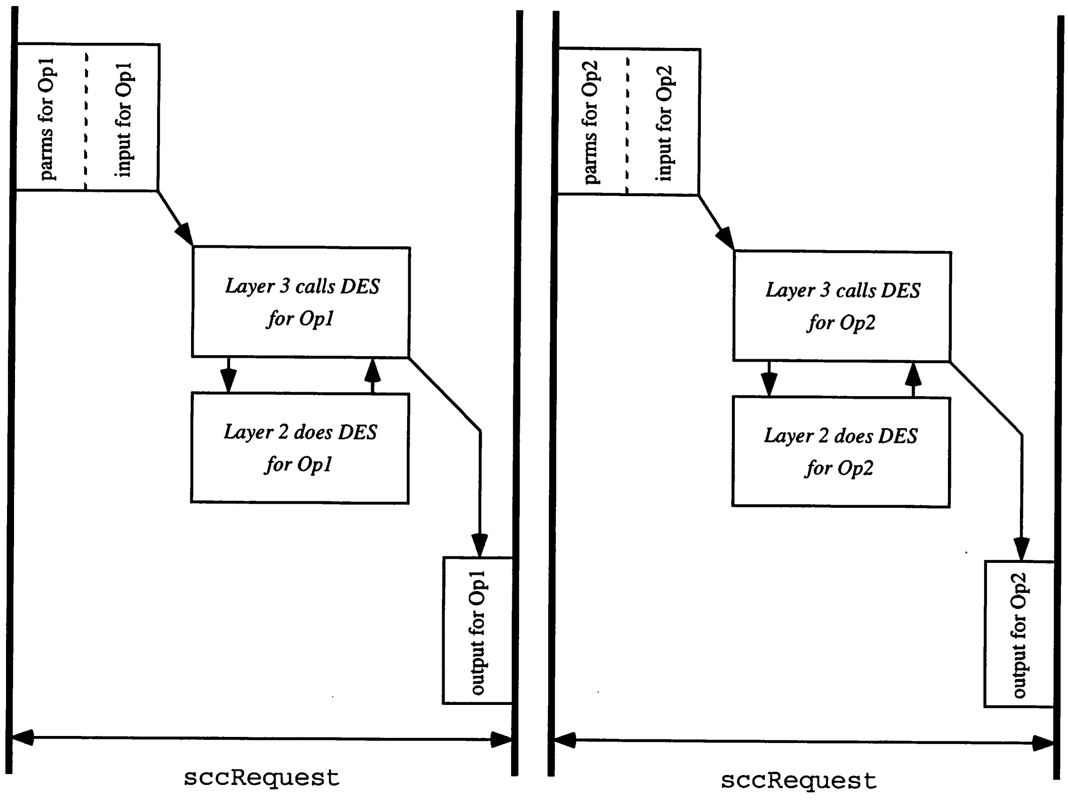


Figure 11 Experiment 1: the application handles each operation as a separate `sccRequest`, with PIO DES.

Results. Speeds ranged from 30 - 290 kilobytes/second.

However, something was still amiss. Each short DES operation requires a minimum number of I/O operations: to set up the DES chip, to get and set up the IV, to get and set up the keys, and then to either drive the data through the chip, or let the FIFO state machine pump it through.

Extrapolating from this back-of-the-envelope sketch to an estimated speed is tricky, due to the complex nature of contemporary CPUs. However, the sketch suggested that multi-megabyte speeds should be possible.

Experiment 5: Reducing Data Transfers

Idea. From our above analysis of what's "minimally necessary" for short-DES, we realized that we were wasting a lot of time with parameter and data transport. In practice, each byte of cipherkey, IV, and data was being handled many times. The bytes came in via FIFOs and DMA into DRAM with the initial `sccRequest` buffer transfer; the CPU was then taking the bytes out of DRAM and putting them into the DES chip; the CPU then took the data out of the DES chip and put it back into DRAM; the CPU then sent the data back to the host through the FIFOs.

However, in theory, each parameter (key, IV, and direction) should require only one transfer: the CPU reads it from the FIFO, then acts. If we let the FIFO state machine pump the data bytes through DES in bulk mode, then the CPU never need handle the data bytes at all.

Experiment. Our next sequence of experiments focused on trying to reduce the number of transfers down to this minimal level.

To simplify things (and since we were starting to try to converge to a "fast short-DES" API), we decided to eliminate key unpacking as a built-in part of the API—since each application has their own way of doing unpacking anyway, and the cost impact was small (for operation sequences distributed over a small number of keys, as we had assumed). Instead, we assumed that, within each application, some "initialization" step would conclude with a plaintext key-table resident in device DRAM. We also decided to standardize operation

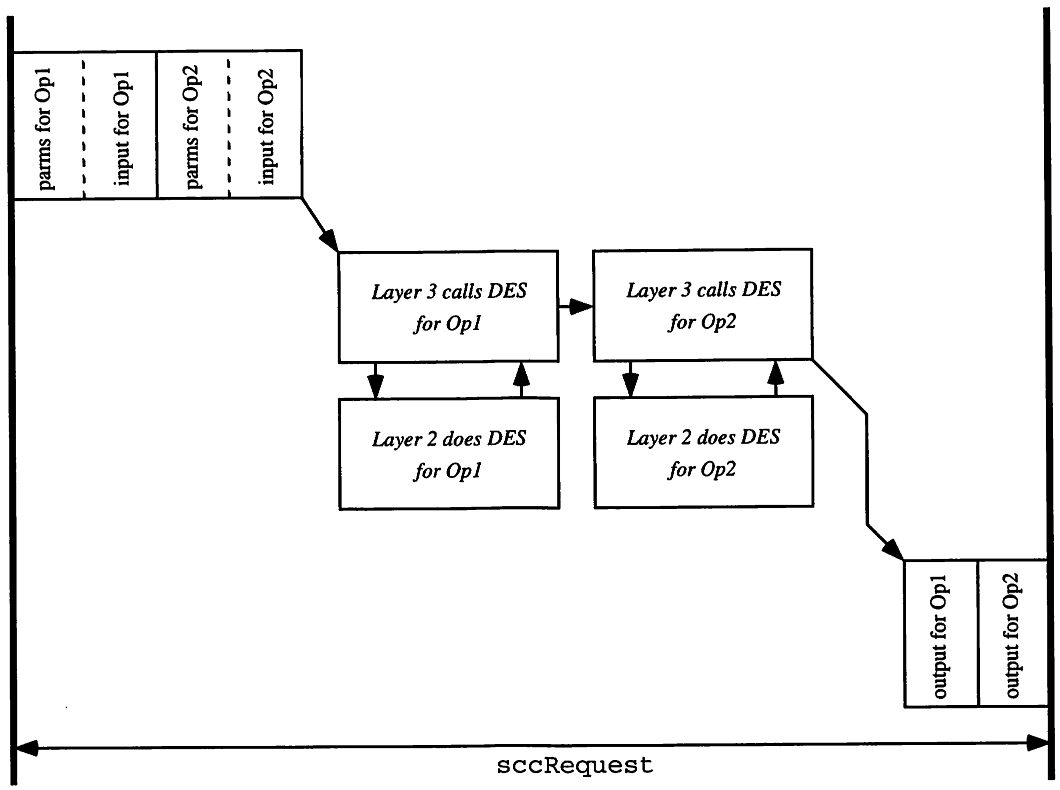


Figure 12 Exp 2: we reduced host-card interaction by batching all the operations into a single `sccRequest`.

lengths to 40 bytes (which, in theory, should mean that the speeds our colleague would see will be 10% higher than our measurements).

We rewrote our host application to generate sequences of requests that each include an index into the internal key-table, instead of a cipherkey. Our card-side application now calls the modified DES Manager (and makes the key table available to it), rather than immediately bringing the request sequence from the PCI Input FIFO into DRAM. For each operation, the modified DES Manager then: resets the DES chip; reads the IV and loads it into the chip; reads (and sanity checks) the key index, looks up the key, and loads it into the chip; reads the data length for this operation; then sets up the state machine to crank that number of bytes through the input FIFOs into the DES chip then back out the output FIFOs.

Figure 14 shows this operation.

Results. Speeds now ranged up to 1400 kilobytes/second.

Experiment 6: Using Memory Mapped I/O

The approach of Experiment 5 showed a major improvement, but performance was still lagging behind what we projected as possible.

Idea. Upon further investigation, we discovered that, in our device, I/O operation speed is not limited by the CPU speed but by the internal ISA bus (effective transfer speed of 8 megabytes/second) When we calculated the number of fetch-and-store transfers necessary for each operation (irrespective of the data length), the slow ISA speed was the bottleneck.

Consequent discussions with the hardware engineers revealed that every I/O register we needed to access—except for the PCI FIFOs—was available from a location that was also memory-mapped—and memory-mapped I/O operations should not be subject to the ISA speed limitations.

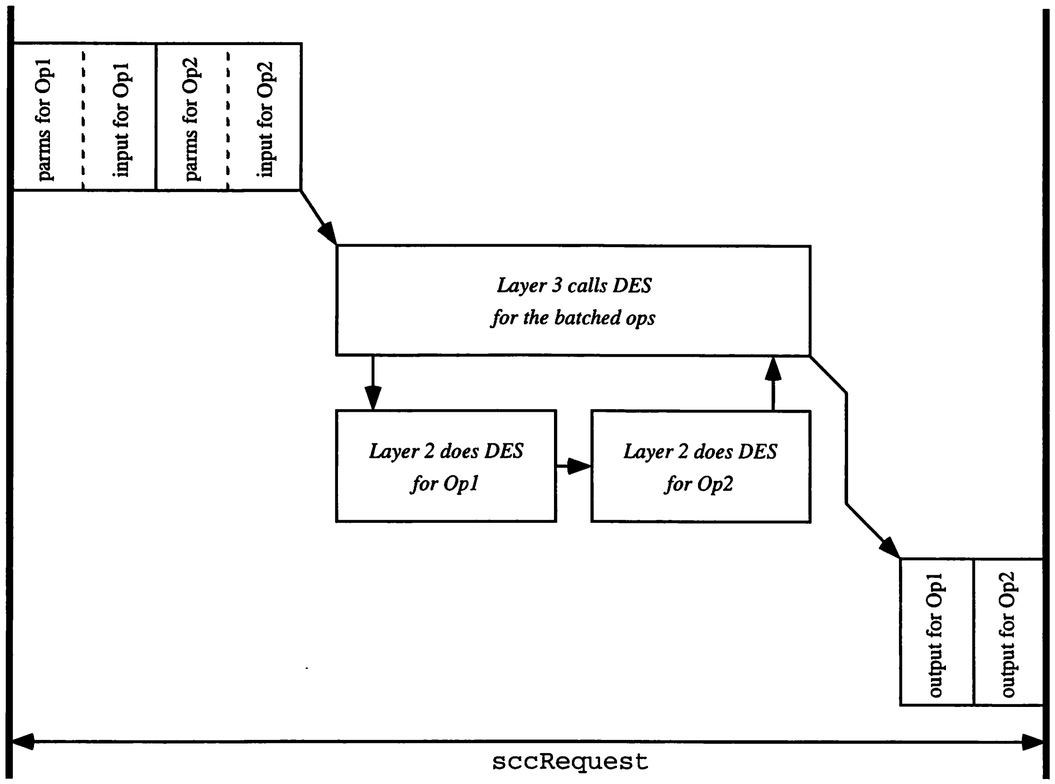


Figure 13 Exp 4: We reduced internal context switches by batching all the operations into a single call to a modified DES Manager in Layer 2.

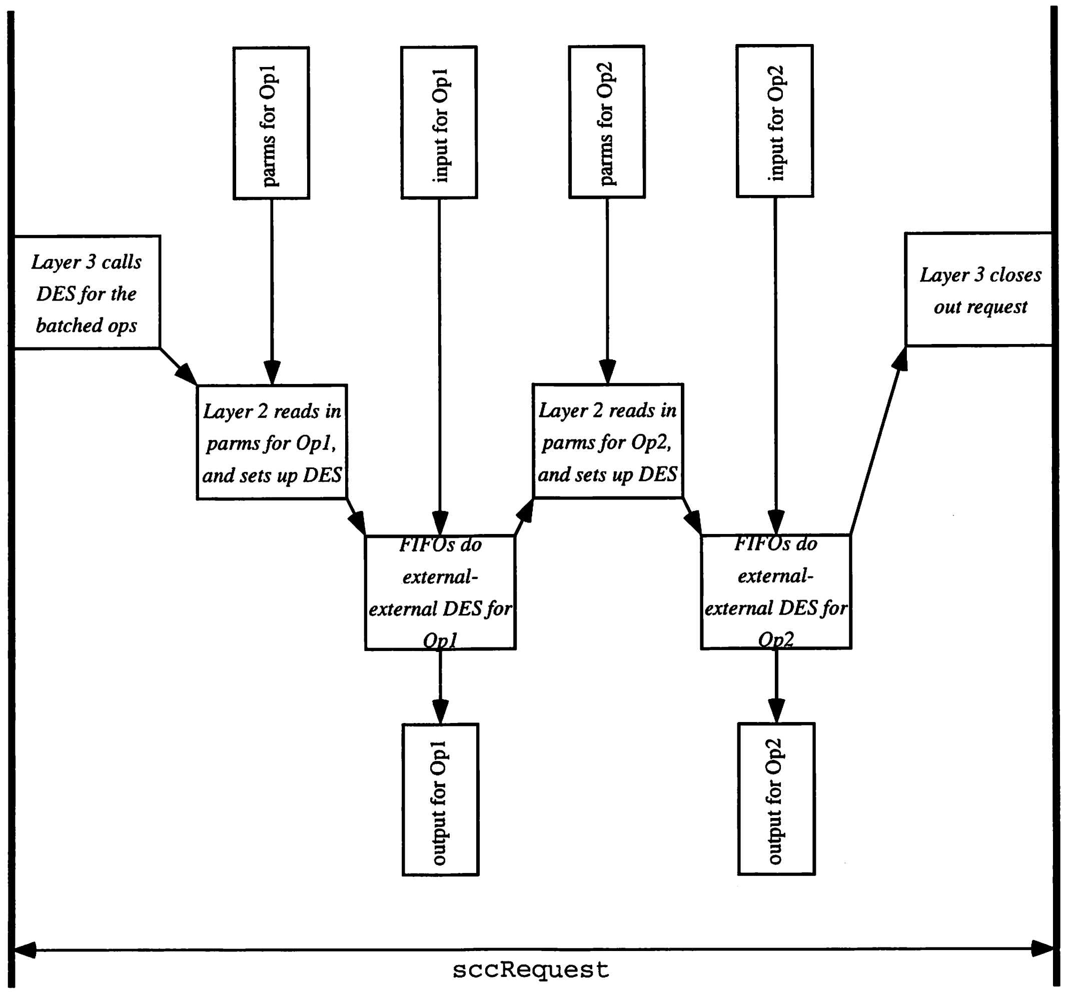

parms for Op1
input for Op1
parms for Op2
input for Op2
Layer 3 calls DES for the batched ops
Layer 3 closes out request
Layer 2 reads in parms for Op1, and sets up DES
Layer 2 reads in parms for Op2, and sets up DES
FIFOs do external-external DES for Op1
FIFOs do external-external DES for Op2
output for Op1
output for Op2
sccRequest

Figure 14 Experiment 5, Experiment 6: We reduce unnecessary data transfers by having the modified DES Manager, for each operation, read in the parameters and configure the FIFOs to do DES directly from and back to the host.

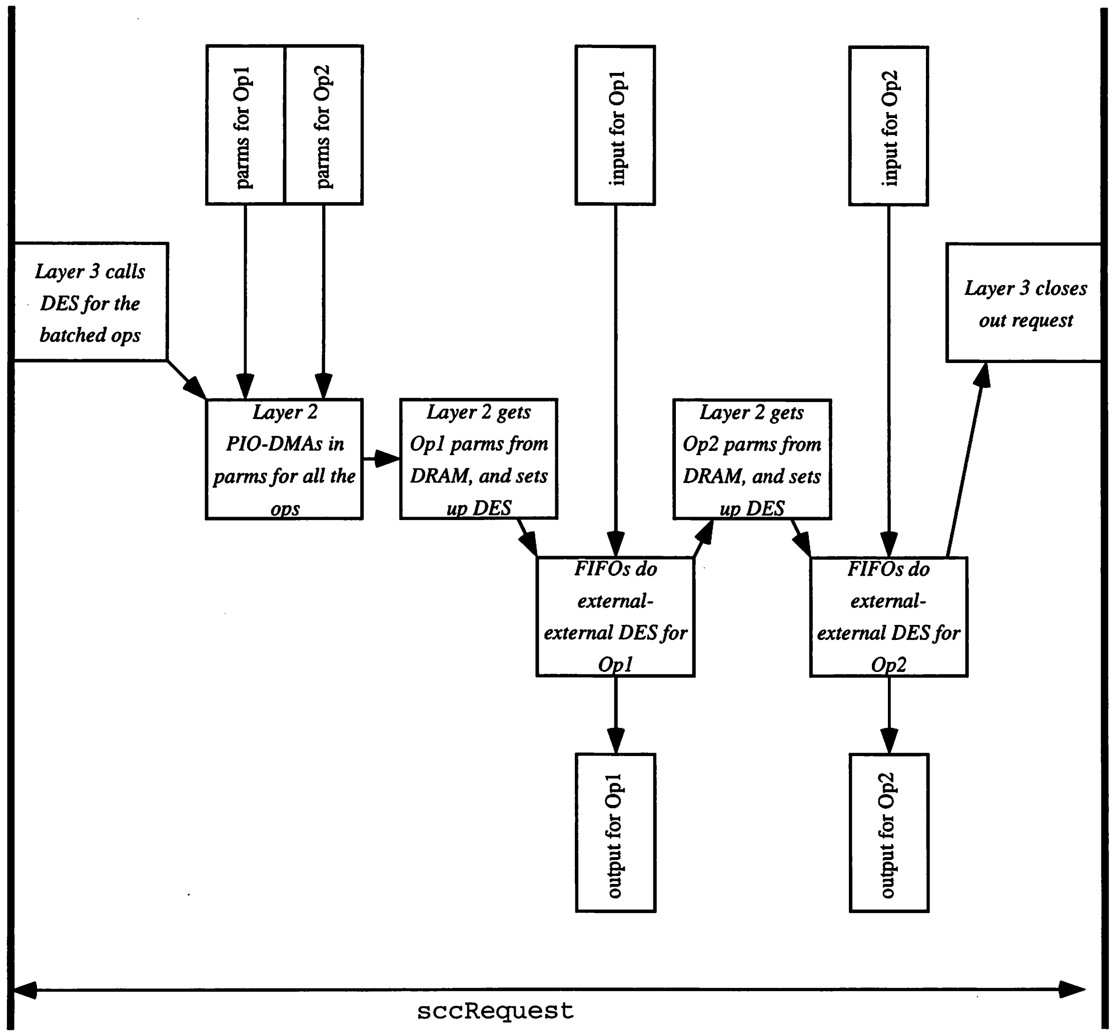

parms for Op1
parms for Op2
input for Op1
input for Op2
Layer 3 calls DES for the batched ops
Layer 3 closes out request
Layer 2 PIO-DMAs in parms for all the ops
Layer 2 gets Op1 parms from DRAM, and sets up DES
Layer 2 gets Op2 parms from DRAM, and sets up DES
FIFOs do external-external DES for Op1
FIFOs do external-external DES for Op2
output for Op1
output for Op2
sccRequest

Figure 15 Experiment 7, Experiment 8: We reduce slow ISA I/Os by batching the parameters for all the operations into one block, and bringing them via PIO DMA.

Experiment. First, we proved the ISA-bottleneck hypothesis by doubling the number of ISA I/O instructions and observing an appropriate halving of the throughput.

Then, we re-worked the modified DES manager of Experiment 5 to use memory-mapped I/O instead of ISA I/O wherever possible. As an unexpected consequence, we discovered a hardware bug—certain state machine polling intermittently caused spurious FIFO reads. (Again, Figure 14 shows this operation.)

Results. Modifying our software again to work around this bug, we measured speeds up to 2500 kilobytes/second.

Experiment 7: Batching Operation Parameters

Idea. The approach of Experiment 6 still requires reading the the per-operation parameters via slow ISA I/O from the PCI Input FIFO. (Reading them via memory-mapped I/O from the Internal Input FIFO is not possible, since we would lose flow control in non-bulk mode.)

However, if we batched the parameters together, we could read them via memory-mapped operations, then change the FIFO configuration, and process the data.

Experiment. In our most recent experiment, we rewrote the host application to batch all the per-operation parameters into one group, prepended to the input data. The modified DES manager then: sets up the Internal FIFOs and the state-machine to read the batched parameters, by-passing the DES chip; reads the batched parameters via memory-mapped operations from the Internal Output FIFO into DRAM; reconfigures the FIFOs; using the buffered parameters, sets up the state-machine and the DES chip to pump each operation's data from the input FIFOs, through DES, then back out the output FIFOs. Figure 15 shows this operation.

Results. With this final approach, we measured speeds approaching 5000 kilobytes/second.

(As a control, we tried this batched-parameters approach using DMA and a separate request buffer, but obtained speeds slightly slower than Experiment 6.)

Experiment 8: Checking the Results

Idea. The results of Experiment 7 pleased us. However, colleagues disrupted this pleasure by pointing out that a recent errata sheet for our DES chip noted that using memory-mapped access for the IV and data length registers may cause incorrect results.

We were tempted to dismiss this news, since the external colleague had merely asked for *fast* cryptography; he said nothing about *correctness*. But we investigated nonetheless.

Experiment. First, we did a known-answer DES test on the implementation of Experiment 7—and it failed. So, we revised that implementation to ensure that the IV and data length registers were access via the slower ISA method. (Again, Figure 15 shows this operation.)

Results. With this final approach, we measured speeds approaching 3000 kilobytes/second.

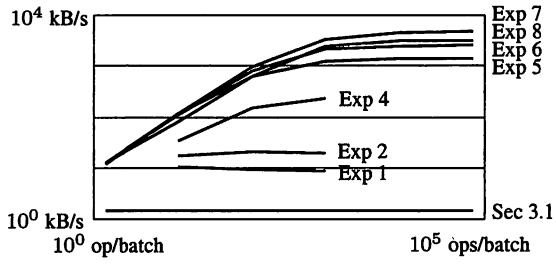


Figure 16 Summary of our short-DES experiments (preliminary figures, on an NT platform)

4 Analysis

4.1 Performance

Figure 16 summarizes the results of our experiment sequence.

On a coarse level, the short-DES speed can be modelled by:

$$\frac{C_1 \cdot Bats + C_2 \cdot Bats \cdot Ops + C_3 \cdot Bats \cdot Ops \cdot DLen}{Bats \cdot Ops \cdot DLen}$$

where *Bats* is the number of host-card batches, *Ops* is the number of operations per batch, *DLen* is the average data length per operation, and C_1, C_2, C_3 are unknown constants, representing the per-batch, per-operation, and per-byte overhead (respectively).

4.1.1 Improving Per-Batch Overhead

The curve of the top traces in Figure 16 suggests that, for fewer than 1000 operations, our speed is still being dominated by the per-batch overhead C_1. To reduce this cost, we are planning another round of hand-tuning the code.

In theory, we could eliminate the per-batch overhead C_1 entirely by modifying the host device driver-Layer 2 interaction to enable indefinite `sccRequests`, with some additional polling or signalling to indicate when more data is ready for transfer. However, our experiments were constrained by the limited resources of our own time, and the constraint that (should the results prove commercially viable) it would be possible to migrate our changes into the commercial offering with a minimum number of component changes. Both of these constraints have prevented us from exploring changes to the device driver protocol at this time.

4.1.2 Improving Per-Operation Overhead

The limitation of short DES puts an upper bound on *DLen*, which suggests a minimum $C_2/DLen$ component that we can never overcome.

API Approaches. For future work, we have been considering various ways to reduce the per-operation overhead C_2 by minimizing the number of per-operation parameter transfers. For example:

- The host application might, within a batch of operations, interleave "parameter blocks" that assert things like "the next N operations all use this key." This eliminates bringing in (and reading) the key index each time.
- The host application itself might process the IVs before or after transmitting the data to the card, as appropriate. (This is not a security issue if the host application already is trusted to provide the IVs.) This eliminates bringing in the IVs, and (since the DES chip has a default IV of zeros after reset) eliminates loading the IVs as well.

However, these approaches have two significant drawbacks. One is the fact that the "short-DES API" (that might eventually emerge in production code) would look less and less like standard DES. Another is that these variations make it much more complicated to benchmark performance meaningfully. How much work should the host application be expected to do? (Remember that the host CPU is probably capable of much greater computational power than the coprocessor CPU.) How do we quantify the "typical request sequences" for which these approaches are tuned, in a manner that enables a potential end user to make meaningful performance predictions?

Hardware Approaches. Another avenue (albeit a long-term one) for reducing per-operation overhead would be to re-design the FIFOs and the state machine.

In hindsight, we can now see that the current hardware has the potential for a fundamental improvement. Currently, the acceleration hardware provides a way to move the *data* very quickly through the engine, but not the *operational parameters*. If the DES engine (or whatever other algorithm engine is being driven this way) expected its data-input to include parameters (e.g., "do the next 40 bytes with key #7 and this IV") interleaved with data, then the per-operation overhead C_2 could approach the per-byte overhead C_3.

The state machine (or whatever system is driving the data through the engine) would need to handle the fact that the number of output bytes may be less than the number of input bytes (since those include the parameters). We also need a way for the CPU to control or restrict the class of engine operations over which the parameters, possibly chosen externally, are allowed to range. For example:

- The external entity may be allowed only to choose certain types of encryption operations (restriction on type).
- The CPU may wish to insert *indirection* on the parameters the external entity chooses and the parameters the engine sees (e.g., the external entity provides an index into an internal table, as we did with keys in the experiments).

The issues of Section 4.2 also apply here.

4.1.3 Improving Per-Byte Overhead

Well-established folklore teaches that the per-byte overhead C_3 is small. Consequently, we doubt C_3 can be improved much, nor that it is significant.

4.2 API Design Issues

Cryptographic APIs, once defined, may appear obvious. But as noted earlier, an implicit goal of this work was, if we were able to substantially improve short-DES performance, to produce a prototype of a new feature that could be migrated into the current commercial offering.

How to design a short-DES API that could provide this superior performance, be usable by a wide range of applications, and be reasonably easy to implement and maintain, raises a number of interesting challenges, including:

- **Key Unpacking.** What is the most general way to handle, in a Layer 2 API, the loading of keys from outside? Each application has its own method (and we haven't even discussed the implications of things such as FIPS 140-1 [6]).
- **Operation Restrictions.** One of the benefits of secure crypto coprocessors is increased security for sensitive operations and data, as well as cryptographic acceleration (which is not necessarily associated with secure coprocessors). Many applications that could use high-speed short DES might want to greatly restrict the modes or keys or IVs or other such parameters that an untrusted host-side entity could choose. How do we handle this in an API?
- **Algorithm Mix.** These techniques could also speed up TDES, SHA-1, DES-MAC, and other algorithms. Which would application programmers require? Would they require operations for different algorithms within the same batch? If so, how do we handle items such as key tables for different algorithms? (For example, allowing the user to choose single-DES operations using parts of TDES keys is risky.) What about variations such as decryption with one key and re-encryption with another, without the plaintext ever leaving the secure boundary? (This last option could speed Kerberos server implementations. [3])
- **Operation Sequences.** As speculated earlier, having the host sort operations in various ways could help speed performance—for some approaches. What's a reasonable balance between full flexibility and manageable implementation?
- **Source/Destination.** These experiments all dealt with operations whose parameters and input were coming from the outside, and whose output was going back to the outside. However, each of these three elements (parameters, input, and output) could also come from inside—and if we start thinking about various types of parameters, the option space grows considerably beyond even this 2^3. What's reasonable?
- **IVs.** Our colleague wanted to choose his own IVs. Some applications would require random IVs (which our device could generate itself); for other applications, the plaintext key (a sensitive item) is re-used for the IV. How do we handle this?
- **Dependent Operations.** Plausible scenarios can constructed for having later operations in a batch use data that resulted from earlier operations—for example, key-unwrapping operations could themselves be included in the same batch as the operations which use these keys. How do we handle this?

As we explore the design space, we are faced with another conundrum. If optimizing performance requires coding a tight CPU loop, then either

- our tight loop will be slowed by n tests (for the n options), or
- or we must implement 2^n different loops—one for each possible set of option choices.

(We briefly considered even having the DES Manager write the loop each time through.) This conundrum faded, however, when it became apparent that CPU speed was not the primary bottleneck.

5 Conclusions

From this experience, we learned many things.

- Meaningful benchmarks of symmetric crypto performance should include data lengths.

- Neglecting to consider how *operational parameters* can be efficiently sent into a cryptographic system can greatly hinder performance—and reduce the benefits of engineering a high-speed data path.
- Neglecting to consider how software can actually use new cryptographic hardware designs can reduce the benefits of these new designs.
- Complex accelerator architectures can hide bottlenecks that are not initially apparent.
- But with a programmable device, software experiments can identify these bottlenecks and overcome many of them.

In the hindsight, an appropriately specified goal ("fast short DES") could have led to an appropriate software and hardware model (e.g., based on standard principles of performance analysis [7]), and thus enabled examination of these issues before the hardware design had even begun. However, one of the contributions of our work is providing this hindsight: in the pressure of product development, hardware tends to be frozen early; and our field tends to introduce a separation between software design and hardware design that prevents a full examination of the interactions.

In future work, we plan to finalize a proposed short-DES API and and attempt to migrate it into the commercial offering (where it can then actually speed real customer applications); we also hope to examine other cryptographic services our device offers, to see if similar techniques will improve performance there. It also would be interesting to explore performance tradeoffs between host-only and coprocessor-enhanced DES for short operations, and then re-examine the security tradeoffs in light of this information.

Furthermore, we hope to use some of our experience in accelerating DES variants to build high-performance prototypes of alternative cryptographic coprocessor applications (such as root-secure private information servers [8], noted earlier, and authenticated encryption [4]).

Availability

Contact Ron Perez (ronpz@us.ibm.com) at IBM T.J. Watson Research Center for current information on the external availability of the experimental code discussed in this paper.

Acknowledgments

The authors gratefully acknowledge helpful discussions with Enzo Condorelli, Joan Dyer, Juan Gonzalez, Ulf Mattsson, Elaine Palmer, and Ron Perez; and also appreciate the helpful comments of the referees.

References

[1] D.G. Abraham, G.M. Dolan, G.P. Double, J.V. Stevens. "Transaction Security Systems." *IBM Systems Journal.* 30: 206-229. 1991.

[2] J. Dyer, R. Perez, S.W. Smith, M. Lindemann. "Application Support Architecture for a High-Performance, Programmable Secure Coprocessor." *22nd National Information Systems Security Conference.* October 1999.

[3] N. Itoi. "Secure Coprocessor Integration with Kerberos V5." *USENIX Security Symposium 2000.*

[4] C. S. Jutla. *Encryption Modes with Almost Free Message Integrity.* Draft Research Report, IBM T.J. Watson Research Center, July 2000.

[5] U. Mattsson, Personal communication, Protegrity Inc. Publication *Performance Report on Secure Coprocessors*, 1999.

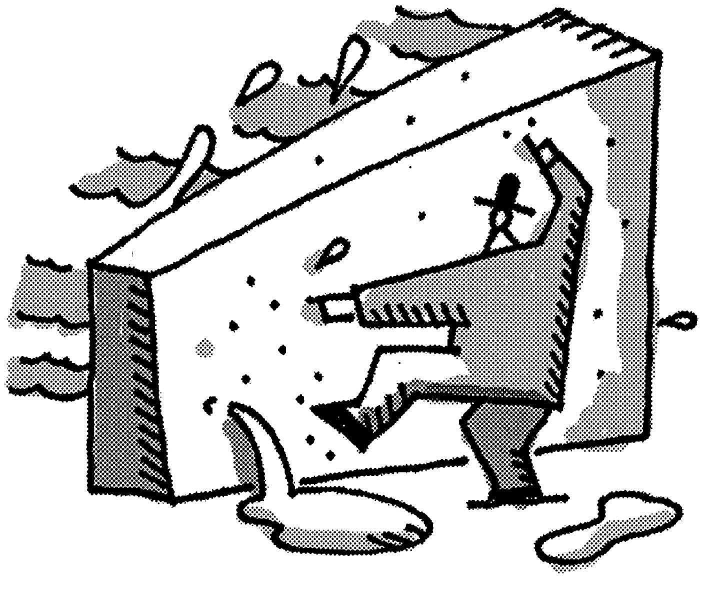
[6] National Institute of Standards and Technology. *Security Requirements for Cryptographic Modules.* Federal Information Processing Standards Publication 140-1, 1994.

[7] C. Smith. *Performance Engineering of Software Systems.* Addison-Wesley, 1990.

[8] S.W. Smith, D. Safford. "Practical Server Privacy Using Secure Coprocessors." *IBM Systems Journal*, to appear.

[9] S.W. Smith, S.H. Weingart. "Building a High-Performance, Programmable Secure Coprocessor." *Computer Networks (Special Issue on Computer Network Security.)* 31: 831-860. April 1999.

FIREWALLS/INTRUSION DETECTION



Session Chair: Mudge, *@stake*

Architecting the Lumeta Firewall Analyzer

Avishai Wool
Lumeta Corporation,
220 Davidson Ave, Somerset NJ 08873
E-mail: `yash@acm.org`.

Abstract

Practically every corporation that is connected to the Internet has at least one firewall, and often many more. However, the protection that these firewalls provide is only as good as the policy they are configured to implement. Therefore, testing, auditing, or reverse-engineering existing firewall configurations should be important components of every corporation's network security practice. Unfortunately, this is easier said than done. Firewall configuration files are written in notoriously hard to read languages, using vendor-specific GUIs. A tool that is sorely missing in the arsenal of firewall administrators and auditors is one that will allow them to analyze the policy on a firewall.

The first passive, analytical, firewall analysis system was the Fang prototype system [MWZ00]. This was the starting point for the new Lumeta Firewall Analyzer (LFA) system. LFA improves upon Fang in many ways. The most significant improvements are that human interaction is limited to providing the firewall configuration, and that LFA automatically issues the "interesting" queries and displays the outputs of all of them, in a way that highlights the risks without cluttering the high-level view. This solves a major usability problem we found with Fang, namely, that users do not know which queries to issue.

The input to the LFA consists of the firewall's routing table, and the firewall's configuration files. The LFA parses these various low-level, vendor-specific, files, and simulates the firewall's behavior against all the packets it could possibly receive. The simulation is done completely offline, without sending any packets. The administrator gets a comprehensive report showing which types of traffic the firewall allows to enter from the Internet into the customer's intranet and which types of traffic are allowed out of the intranet. The LFA's report is presented as a set of explicit web pages, which are rich with links and cross references to further detail (allowing for easy drill-down). This paper describes the design and architecture of the LFA.

1 Introduction

1.1 Background

Firewalls are the cornerstones of corporate intranet security. Once a firewall is acquired, a security/systems administrator has to configure and manage it to realize an appropriate security policy for the particular needs of the company. This is a crucial task; quoting [RGR97]: "The single most important factor of your firewall's security is how you configure it".

Even understanding the deployed firewall policy can be a daunting task. Administrators today have no easy way of answering questions such as "can I telnet from here to there?", or "from which machines can our DMZ be reached, and with which services?", or "what will be the effect of adding this rule to the firewall?". These are basic questions that administrators need to answer regularly in order to perform their jobs, and sometimes more importantly, in order to explain the policy and its consequences to their management. There are several reasons why this task is difficult, for instance:

(i) Firewall configuration languages tend to be arcane, very low level, and highly vendor specific.

(ii) Vendor-supplied GUIs require their users to click through several windows in order to fully understand even a single rule: at a minimum, the user needs to check the IP addresses of the source and destination fields, and the protocols and ports underlying the service field.

(iii) Firewall rule-bases are sensitive to rule order. Several rules may match a particular packet, and usually the first matching rule is applied – so changing the rule order, or inserting a correct rule in the wrong place, may lead to unexpected behavior and possible security breaches.

(iv) Alternating PASS and DROP rules create rule-bases that have complex interactions between different rules. What policy such a rule-base is enforcing is hard for humans to comprehend when there are more than a handful of rules.

A tool that is sorely missing in the arsenal of firewall administrators and auditors is one that will allow them to analyze, test, debug, or reverse-engineer the policy on a firewall. Such a tool needs to be exhaustive in its coverage, be high level, and be convenient to use. This paper describes the evolution and architecture of the Lumeta Firewall Analyzer (LFA), a second generation system that addresses the analysis needs of firewall administrators, security consultants, and auditors.

1.2 The Fang System

The first passive, analytical, firewall analysis system was the Fang prototype system [MWZ00]. Fang read all the vendor-specific configuration files, and built an internal representation of the implied policy. It provided a graphical user interface (GUI) for posing queries of the form "does the policy allow service S from A to B?". Fang would then simulate the firewall's policy against the query, and display the results back onto the user's screen.

Before Fang could be used, it needed to have an instantiated model of the firewall connectivity, which contained details like how many interfaces the firewall has, which subnets are connected to each interface, and where the Internet is situated with respect to the firewall. Therefore, before querying the firewall policy, a Fang user needed to write a firewall connectivity description file. The language used to describe the firewall connectivity was derived from the Firmato MDL language [BMNW99].

The core of Fang's query engine was a combination of a graph algorithm and a rule-base simulator. It took as input a user query consisting of source and destination host-groups (arbitrary sets of IP addresses, up to a wildcard "all possible IP addresses"), and a service group (up to a wildcard "all possible services"). It would then simulate the behavior of the firewall's rule-base on all the packets described by the query, and compute which portions of the original query would manage to reach from source to destination: Perhaps only a subset of the queried services are allowed, and only between subsets of the specified source and destination host-groups.

1.3 Contributions

To test Fang's usability and the value it provided, we collected feedback from beta testers. This feedback raised issues we needed to address. The new LFA architecture introduces several new features that address these issues:

- The user does not need to write the firewall connectivity file any more. LFA has a new front-end module that takes a formatted routing table and automatically creates the firewall connectivity file.
- Using a GUI as an input mechanism turned out to be difficult for users. Instead, LFA is now a batch process, that simulates the firewall policy against practically every possible packet.
- A crucial part of the batch processing is the automatic selection of queries. Our choice of queries needs to ensure comprehensive coverage, to highlight any risks, and to make sense to users without overwhelming them with minutiae.
- The LFA output is now formatted as a collection of web pages (HTML). This format gives us the ability to present the output at many levels of abstraction and from multiple viewpoints, allowing easy drill-down to details without cluttering the high level view.
- We needed to support more firewall vendors. For this purpose, LFA now uses an intermediate firewall configuration language, to which we convert the various vendors' configurations.

Organization: In Section 2 we describe the components of the LFA architecture and the design decisions that led us to this architecture. In Section 3 we discuss some related work. In Section 4 we provide an annotated example of how the LFA works. We conclude in Section 5.

2 The LFA Architecture

The main contribution of the Fang prototype was its core query engine. The combination of its internal firewall connectivity model, data structures, and efficient algorithms, demonstrated that it is feasible to analytically simulate a firewall's policy offline. However, from the beta-testers' feedback we got, it became apparent that the software architecture needed to be revisited in order to take the core technology from a prototype into a product. In the next sections we describe the problems that we identified in the Fang prototype, and their solutions within the LFA.

2.1 Describing the Firewall Connectivity

As we mentioned above, before using Fang the user needed to write a firewall connectivity description file, using the Firmato MDL language [BMNW99]. For every network interface card (NIC) on the firewall, the firewall connectivity description file contains a list of IP address ranges that are located behind that NIC. These lists are required to be disjoint: each IP address is allowed to appear only once. This requirement is fundamental to the simulation process: For every possible packet, Fang needs to know which firewall interfaces the packet would cross on its path from source to destination—and thereby, which firewall rule-bases would be applied to it.

The need to write a firewall connectivity file caused two problems. First, the user had to learn the syntax and semantics of the MDL language, which takes time and effort. Second, and more important, the information that is needed to describe the firewall connectivity is not readily available to firewall administrators in a suitable format. This information is typically only encoded in the firewall's routing table. However, routing table entries are usually not disjoint: It is common to have many overlapping routing table entries that cover the same IP address. The semantics of a routing table determine which route entry is used for a given IP address: it is the most specific one, i.e., the entry for the smallest subnet that contains the given IP address is the one that determines the route to that IP address. The task of accessing the routing table, and manually converting it into lists of disjoint IP address ranges, turned out to be difficult and error prone.

To solve both problems, the Lumeta Firewall Analyzer introduced a a new front-end module, called `route2hos`, that mechanically converts a routing table into a Firmato MDL firewall connectivity file. All that is required of the user is to provide the firewall's routing table (in the form of the output of the `netstat` command on Unix systems).

The `route2hos` module uses an engine that implements the routing table semantics. In other words, for a given IP address, it is able to determine over which NIC a packet with this address as its destination would be routed. By judiciously using this engine against the subnets listed in the routing table, `route2hos` is able to create the disjoint lists of IP address ranges that the Fang query engine requires. The output of `route2hos` is the firewall connectivity description file, in the MDL language.

As part of the processing done by `route2hos`, it produces definitions for two special host groups, called Inside and Outside. The Outside host group consists of all the IP addresses that get routed via the default interface, according to the firewall's routing table. This host group typically includes the Internet, and any of the corporation's subnets that are external to the firewall. The Inside host group is everything else. These two host groups are later used in the query processing (see below).

2.2 What to Query?

The Fang prototype had a graphical user interface which allowed the user to enter queries of their choice. However, during beta testing we discovered that users do not know which queries they need to try. They were not sure which services are risky, nor which host groups needed to be checked. Furthermore, on a reasonably configured firewall, most queries return uninteresting results, e.g.: "is telnet allowed into my network?"; "No"; etc. This causes users to lose interest and leads to a partial simulation of the policy. Most importantly, the queries that are likely to find the problems in the rule-base are often precisely those queries that the user does not know to try.

To solve these problems, the Lumeta Firewall Analyzer takes the burden of choosing the queries off the user's shoulders. It does this by querying *everything*. In fact, we completely eliminated the GUI as an input mechanism in the LFA, and replaced it by a batch process, which repeatedly calls Fang's query engine.

Clearly, it is impossible to simulate all the packet combinations one by one. Enumerating all the possible combinations of source and destination IP addresses (32 bits each), protocol (8 bits), and source and destination port numbers (16 bits each), gives rise to an enumeration space of 2^{104}.

There are two facts that allow LFA to circumvent this combinatorial explosion: (i) the Fang query engine processes aggregated queries very efficiently, and (ii) after the `route2hos` processing the LFA knows which IP addresses are external to the firewall (this is the Outside host group). Combining these two facts, LFA can issue the query "list the types of traffic that can enter from the Outside to the Inside using any service". We denote such a query by

$$\text{Outside} \to \text{Inside} : *.$$

The result is a list of `(src, dest, srv)` tuples describing the allowed incoming traffic, in which the IP addresses of `src` are contained in the Outside host group, the IP addresses of `dest` are contained in Inside, and the service is `srv`. Similarly, LFA can make the outgoing query "Inside $\to$ Outside : $*$", switching the roles of Inside and Outside.

After experimenting with the approach we just outlined, we discovered that users had difficulty in interpreting its results. For instance, suppose the firewall has a rather typical rule of the form "from anywhere, to my-server, allow any service". The query "Outside $\to$ Inside : $*$" would produce the response "Outside $\to$ my-server : $*$". This response does not convey to the user that "$*$" (any service) includes quite a few high-risk services that should probably not be allowed—if this fact was obvious to the user, he would not have written such a rule in the first place! Users found the results much easier to interpret if instead of presenting a blanket response saying "any service" is allowed, we presented them with a long list of individual services that are allowed.

Therefore, the LFA in fact does not make the query "Outside $\to$ Inside : $*$". Instead it issues a set of focused queries: "Outside $\to$ Inside : `dns`"; "Outside $\to$ Inside : `netbios`"; etc., and similarly for outgoing traffic. The list of services that are queried in this way is made of two parts: a list of well known services, plus a list containing every specific service that appears in some rule on the firewall. We have found that querying individual services this way makes the query results, and the risks they entail, much more explicit. The user has two possible cues indicating risk: (1) If a rule is wide open, there will be a very long list of individual services appearing in the query results (more services == more risk); (2) The user will see services he may either recognize as dangerous, or not recognize at all (making them worrisome).

Note, however, that by querying individual services this way, LFA may miss some services. A service that is not on the LFA's list of "known services", and does not appear explicitly on any rule, will not be queried.

To ensure this does not happen, LFA performs two additional sets of queries. In these queries, the queried service is the "all service" wildcard "*". However, following the same philosophy from before, we attempt to make the queries specific, in a different way. For incoming traffic, LFA makes queries of the form "Outside $\to$ internal-host-group : $*$", where "internal-host-group" goes over every internal host group.[1] LFA then goes over the internal host groups again, making outbound queries of the form "internal-host-group $\to$ Outside : $*$".

The results of all these queries are organized into four reports, called "Analysis by service: Incoming", "Analysis by service: Outgoing", "Analysis by host group: Incoming", and "Analysis by host group: Outgoing". This organization offers the user the opportunity to look at the firewall configuration from different viewpoints, while providing a comprehensive coverage of the traffic the firewall may encounter.

2.3 Supporting Multiple Vendors

The core query engine uses a model of the firewall rule-base, which is generic and vendor-independent. However, in order to instantiate this model, the Lumeta Firewall Analyzer (LFA) needs to be able to parse the vendor-specific configuration files, and if necessary, to convert the vendor's firewall semantics into their equivalent in the LFA model. The Fang prototype provided native support (within the C code implementing the core query engine) only for the Lucent Managed Firewall [LMF99] configuration file syntax.

When we started adding support for other vendors (notably Check Point's and Cisco's products), we decided not to include additional parsers for these vendors' languages within the core. Instead, we opted for an architecture centered around an intermediate language. We chose to write a separate front-end conversion utility for each supported vendor. We chose to write these utilities using the Perl programming language. The front-ends would take the vendor's files and translate them into the LFA's intermediate language. We had three options for an intermediate language. We could base it on an access-control-list language, or on one of Check Point's languages, or on the Lucent Managed Firewall (LMF) language.

Access-control-list languages such as Cisco's IOS [IOS00] and PIX [PIX97] configuration languages, or the Linux `ipchains` (cf. [Rus00]) script language, do not support named host groups, and a rule's source and destination are restricted to be CIDR-block subnets. Therefore, an access-control-list language was deemed too low-level for our purposes; converting other firewall configuration languages to it would lose information and

[1]A host group is considered to be internal if it has a non-empty intersection with in the Inside host group.

greatly increase the configuration size.[2]

Check Point [Che97] uses two separate languages in the configuration of their FW-1 product: the INSPECT language, and the language within the `*.W/*.C` policy files. The INSPECT language does support IP ranges but does not support naming, so it was deemed too low level. The `.W` language does support naming, groups, and ranges, however, it has the opposite problem: it is too expressive. It contains many irrelevant details, such as the colors in which to render the icons on screen, and has a syntax that is much harder to parse or to synthesize.

The language we chose to base our intermediate language on was the LMF configuration language. The basic LMF language is relatively easy to parse and to synthesize, yet contains higher-level constructs such as service groups and host groups, named user-defined services, named host groups, and arbitrary ranges of IP addresses.

Since we only use the language internally, within the LFA, there was no reason to maintain strict compatibility with the real LMF language. Therefore we only used some of the LMF language components and ignored others. Furthermore, we did need to extend the LMF language to incorporate features which LMF itself does not support, such as negated host groups.[3]

2.4 Presentation of Results

In addition to letting the user specify her query, the Fang GUI also displayed the query output to the user. The GUI had a basic mode showing the names of the sources, destinations, and services in the resulting `(src, dest, srv)` tuple. The user had the ability to expand each tuple to show the IP addresses and port numbers (all the components expanded simultaneously). However, beta testers felt that these two display modes were too limiting.

When we discarded the GUI, we needed an alternative mechanism to view the query results. Our choice was to use an HTML-based display. We updated the core query engine so it will dump all its findings into several formatted plain-text output files. Then we created a collection of Perl back-end utilities that convert the output files into a set of web pages.

[2]A single IP address range may need multiple CIDR block subnets to cover it, the worst case being the range 0.0.0.1–255.255.255.254, which requires 62 separate CIDR blocks.

[3]A negated host group is shorthand for the IP addresses that are not contained in the host group.

The back-ends create four support web pages:

Original rules. This page shows the rule-base in a format that is as close as possible to the format used by the vendor's management tools.

Expanded rules. This page shows the rule-base after conversion into the LFA intermediate language.

Services. This page shows a table of all the service definitions (protocols and port numbers), with the containment relationships[4] between services. A service has a hyperlink to every service group containing it, and to every service it contains.

Host groups. This page shows a table of the definitions (IP addresses) of all the host groups encountered in the firewall rule-base, with the containment relationships between host groups represented by hyperlinks.

In addition to the support pages, the back-ends create web pages for the four query reports we mentioned in Section 2.2: Analysis by service (Incoming and Outgoing), and Analysis by host group (Incoming and Outgoing). Each query result tuple is linked to the appropriate entries in the Host groups and Services pages, with a direct link to the Expanded rules page pointing to the rule allowing the traffic through. A typical LFA report contains hundreds of such hyperlinks (depending on the complexity of the rule-base).

Besides the extensive navigation capability offered by the various links, we added a JavaScript-based navigation bar, and JavaScript scrolling functions that highlight the table entries in the Rules, Services, and Host Groups tables.

An advantage of such a web-based display is that it does not impose a reading order on the user, and allows easy access to any level of detail the user desires to view. The query result pages just show the names, and the user can choose whether to drill down on each component.

Section 4 contains excerpts from some of the produced web pages.

2.5 Naming Things

An important part of the Lumeta Firewall Analysis involves assigning names to services and host groups.

[4]A service group s_1 contains service s_2 if the s_2's protocol is one of s_1's protocols, and s_2's port numbers are contained in the range of s_1's port numbers.

For services and service groups, we use several sources of naming information. First, the LFA has a fairly long list of "well known" service definitions. So if the firewall rule-base contains a rule that refers to tcp on port 443, LFA displays it as https. Second, most firewalls have built-in named definitions which we use. Finally, for firewalls that support user-defined services, we read those names in.

If the name and definition we get from two sources both match, we only show the service once. However, sometimes there are mismatches: e.g., Check Point has a predefined service called icmp-proto, which has the same definition as an LFA-defined service called ALL_ICMP. In such cases we incorporate both names into the reports. Another type of mismatch is when the same name is used with different definitions. For instance, there is an LFA-defined service called traceroute, which is defined as udp with a port range of 32000–53000. Check Point has a predefined service with the same name but defined with a port range of 33001–65535. To avoid ambiguity, we prefix the service name with the source of the definition.

For host groups, we rely on the naming information that the firewall provides, which consists of user-defined names. If the firewall does not support host group names (as is the case, e.g., for Cisco IOS [IOS00] access-control-lists), we use the IP addresses themselves as the name. In addition, in all cases, LFA attempts to supplement the host group names with DNS lookups where possible. A reverse DNS lookup is performed for every individual IP address that appears anywhere in the rule-base. For subnets, LFA uses a heuristic to pick a representative IP address in the subnet, and looks up that IP address' name.

2.6 Check Point-Specific Features

The Lumeta Firewall Analyzer (LFA) front-end ckp2lfa, that converts Check Point FW-1 configurations into the LFA intermediate language, has to deal with several Check Point-specific features.

Global properties These are properties which are accessed through a separate tab in Check Point's management module, and are not seen in the rules table shown in the Check Point GUI. Some of the properties control remote management access to the firewall itself, dns access through the firewall, and icmp access. Depending on their setting, these properties in fact create implicit rules that are inserted into the rule-base at certain positions. The ckp2lfa front-end converts these FW-1 properties into explicit rules, and places them in their appropriate position in the rule base (First/Before-Last/Last).

Object groups Check Point FW-1 allows network objects (i.e., host groups) to be defined as groups of other objects, which themselves may be groups, thus creating a containment hierarchy of groups. If the hierarchy is complicated enough, FW-1 users sometimes lose track of what IP addresses the group actually consists of, which leads to all kinds of configuration errors. The ckp2lfa front-end flattens out the hierarchy, by computing the explicit list of IP addresses that belong to such a group object. This flattening does not lose information: one of the features of the LFA query engine is that it computes the host group containment relationships from the IP addresses, regardless of whether a host group was defined as a group or not.

Negated objects Check Point FW-1 allows the firewall administrator to define rules which refer to IP addresses "not in" a host group, or to services "not in" a service group. The ckp2lfa front-end converts the implicit definition into an explicit one, by computing all the IP addresses that do not belong to the negated host group.

3 Related Work

3.1 Active Vulnerability Testing

A number of vulnerability testing tools are available in the market today. Some are commercial, from vendors such as Cisco [CSS00] and ISS [ISS00], others are free such as Fyodor's nmap [Fyo00]. These tools physically connect to the intranet, and probe the network, thereby testing the deployed routing and firewall policies. These tools are *active*: they send packets on the network and diagnose the packets they receive in return. As such, they suffer from several restrictions:

(i) If the intranet is large, with many thousands of machines, testing all of them using an active vulnerability tester is prohibitively slow. Certainly, an active test tool cannot check against every possible combination of source and destination IP address, port numbers and protocols. Hence, users are forced to select which machines should be tested, and hope that the untested machines are secure. Unfortunately, it only takes one vulnerable machine to allow a penetration.

(ii) Vulnerability testing tools can only catch one type of firewall configuration error: allowing unauthorized packets through. They do not catch the other type of error: inadvertently blocking authorized packets. This second type of error is typically detected by a "deploy and wait for complaints" strategy, which is disruptive to the network users and may cut off critical business applications.

(iii) Active testing is always after-the-fact. Detecting a problem after the new policy has been deployed is dangerous (the network is vulnerable until the problem is detected and a safe policy is deployed), costly (deploying policy in a large network is a time consuming and error prone job), and disruptive to users. Having the ability to cold-test the policy before deploying it is a big improvement.

(iv) An active vulnerability tester sends packets, and detects problems by examining the return packets it gets or doesn't get. Therefore, it is inherently unable to test network's vulnerability to spoofing attacks: If the tester would spoof the source IP address on the packets it sends, it would never receive any return packets, and will have no indication whether the spoofed packets reach their destination or not.

(v) An active tester can only test from its physical location in the network topology. A problem that is specific to a path through the network that does not involve the host on which the active tool is running will go undetected.

3.2 Distributed Firewalls

Recently there has been a renewed interest in firewall research, focusing on Bellovin's idea of a distributed firewall [Bel99]. A working prototype has been developed under OpenBSD [IKBS00]. The basic idea is to make every host into a firewall that filters traffic to and from itself. This trend is growing in the commercial world as well: personal firewalls for PCs, such as Zone Labs [Zon00] and BlackICE [Bla00], are becoming more common, as high-bandwidth, always-on, Internet connections like DSL and Cable become more widespread.

The main advantages of a distributed firewall are that (i) since the filtering is at the endpoint, it can be based on more detailed information (such as the binary executable that is sending or receiving the packets); and (ii) there is no bandwidth bottleneck at the perimeter firewall. The main difficulties with a distributed firewall are (i) the need for a central policy to control the filtering, and (ii) the need to ensure that *every* device in the network is protected, including infrastructure devices like routers and printers.

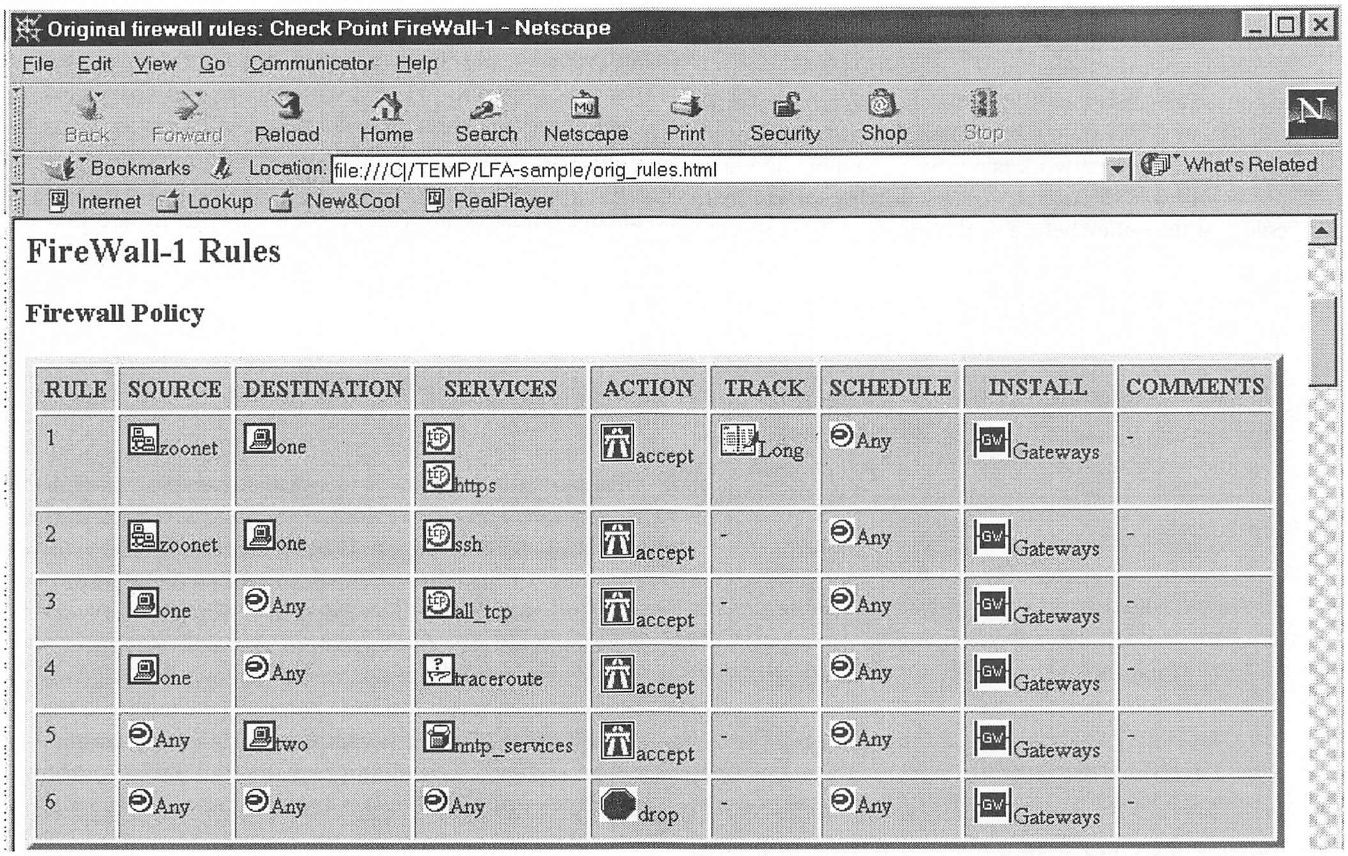
It is this author's opinion that a distributed firewall architecture will augment, rather than replace, the perimeter firewall. The conventional firewall will remain as an enterprise network's first line of defense. The fact that one can put a lock on every office door does not make the guard at the building entrance unnecessary: there is still valuable stuff in the hallways, and not everyone uses the lock properly. When a widely deployed distributed firewall system becomes available, it will most likely be used as a second line of defense, behind the perimeter firewall. The perimeter firewall will continue to protect all the infrastructure that is not controlled by the new architecture, to defend against denial-of-service attacks, and to ensure central control.

4 An Example

In this section we show an annotated example which illustrates the flow of data through the various components of the LFA. This example is based upon a firewall rule-base that was installed on a real firewall protecting a production network of a large enterprise. Using the LFA report, the firewall's administrators were able to correct a major security risk that was present in their firewall configuration. For demonstration purposes, we recreated the key elements of that risky configuration onto a lab machine, and ran the resulting files through the LFA. The report excerpt shown here is from the lab machine. The full web-based sample report is available online from [Lum01].

In Figure 1 we see a web page showing a Check Point FW-1 rule-base. This is the LFA's starting point. The only processing that was done to create this page was to convert Check Point's configuration files into HTML, rendered in a format that is quite close to that of FW-1's management module (down to the level of user-defined colors for various objects). The conversion utility we used is an improved version of the `fwrules50` program [XOS+00].

At a cursory glance, the rule-base looks rather simple, protecting two machines (called `one` and `two`). Machine `one` seems to be a web server, and machine `two` seems to be a Usenet (nntp) news server. The policy is


Original firewall rules: Check Point FireWall-1 - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Location: file:///C|/TEMP/LFA-sample/orig_rules.html
What's Related
Internet Lookup New&Cool RealPlayer

FireWall-1 Rules

Firewall Policy

RULE	SOURCE	DESTINATION	SERVICES	ACTION	TRACK	SCHEDULE	INSTALL	COMMENTS
1	zoonet	one	http https	accept	Long	Any	Gateways	-
2	zoonet	one	ssh	accept	-	Any	Gateways	-
3	one	Any	all_tcp	accept	-	Any	Gateways	-
4	one	Any	traceroute	accept	-	Any	Gateways	-
5	Any	two	nntp_services	accept	-	Any	Gateways	-
6	Any	Any	Any	drop	-	Any	Gateways	-

Figure 1: The original rule-base, rendered in HTML.

quite lax on outbound services (rule 3 allows all types of tcp outbound), but seems quite reasonable for inbound connections, allowing only http, https, ssh, and nntp.

In Figure 2 we can see an HTML rendering, produced for the same rule-base, of the LFA's intermediate language, as discussed in Section 2.3. The figure shows the results of the Check-Point-to-LFA front-end conversion utility, ckp2lfa, post-processed into an HTML-based report (called the Expanded Rules report) by the back-end utilities.

We can see that the rule-base now has several additional rules. These rules are derived from Check Point "properties", which are controlled through a separate tab in Check Point's management module. The properties that are selected by the administrator create implicit rules that are inserted into the rule-base at certain positions. One of the tasks of the ckp2lfa front-end is to convert all these implicit rules into their explicit equivalents, and insert them in their correct positions in the rule-base.

Figure 2 shows the effects of properties that govern DNS and ICMP traffic, and of the property that controls remote management access to the firewall itself. After ckp2lfa converts the implicit rules into explicit ones, we can see that rules 1, 3, and 10, are wide open (allowing traffic from anywhere to anywhere). Unfortunately, these rules represent the effects of Check Point FW-1's default settings. Based on client configuration files we have seen, leaving these properties at their default setting seems to be a common mistake among FW-1 administrators.

Another piece of information that is clear after the ckp2lfa conversion is that the firewall is actually performing Network Address Translation (NAT) on the address on machine one: Rules 4–8 show that machine one has both a valid (routable) IP address and a private IP address. The firewall translates between the two addresses based on the direction of the packets.

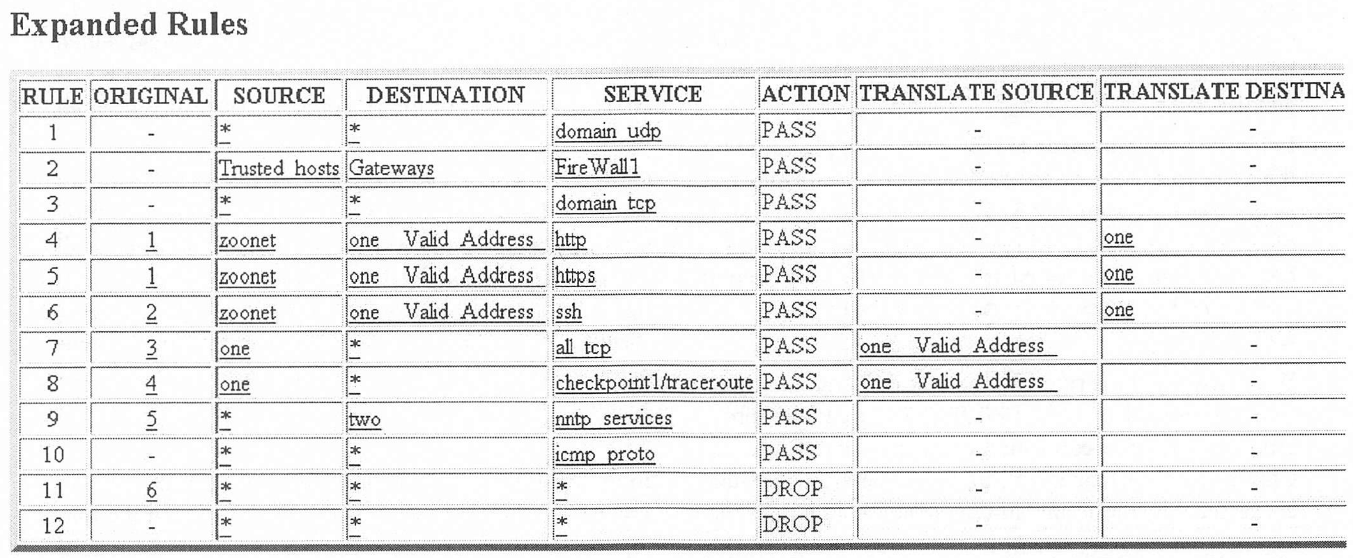
The next step in the processing is the the route2hos front-end, which converts the firewall's routing table into a Firmato MDL firewall connectivity file. Instead of showing the firewall connectivity file itself, in Figure 3 we show a graphical representation of the firewall connectivity, which is derived from the MDL firewall connectivity file using the graph visualization tool dot [GKNV93] [Dot01]. We emphasize that Figure 3 is completely machine-generated, with no manual tweaking. The figure shows the IP addresses behind each of the firewall's three internal interfaces. We can see that interface if_2 is connected to an RFC 1918 private IP address subnet, with a single routable IP address added (this is the valid IP address of machine one, which is NATed). The rest of the IP address space, including all of the Internet, is behind interface if_0.

Once the Check Point configuration files have been converted to the LFA intermediate language, and the routing table has been converted into an MDL network firewall connectivity file, the LFA proceeds to simulate the configured policy. This is done by the core query engine (Section 2.2). The output of the core engine is then rendered in HTML by the back-end utilities, which also create all the cross-links between various components of the report.

In Figure 4 we see a portion of the "Analysis by service: Incoming" HTML-based report, which is one of the four reports that LFA creates. The figure shows the results of the query "Outside $\rightarrow$ Inside : netbios", meaning "Can netbios traffic cross the firewall from the Outside to the Inside?".

Somewhat surprisingly, the report shows that netbios traffic is allowed from anywhere on the Outside, to machine two. The figure shows the user-defined name ("two") alongside the result of a reverse dns lookup on the IP address of that machine (recall Section 2.5). We can see in the figure that the culprit rule which allows netbios traffic through is rule number 9. All the underlined values shown in Figure 4 are hyperlinks. Clicking on the "9" link brings the user to the Expanded Rules report (recall Figure 2), with rule 9 highlighted. Looking back at Figure 2, we see that rule 9 indeed refers to machine two, however, the service listed is called nntp_services, not netbios.

Clicking on the nntp_services link from the Expanded Rules report (Figure 2) brings the user to the Services report, the relevant portion of which is shown in Figure 5. We can see that the definition of nntp_services has two components: one with tcp on destination port 119 (this is the correct definition), and one with tcp on *source* port 119. The latter definition is very risky and is the cause for netbios (and, indeed, any other tcp service) being allowed through the firewall. This is since the choice of source port is completely under the control of the sender of the packet. There is nothing to prevent an attacker from setting the source port to 119 and the destination port to 139 (netbios): the firewall would let the packet through based on its source port, and allow it to access the netbios port on the target machine. This is actually part of a hacking



Expanded Rules

RULE	ORIGINAL	SOURCE	DESTINATION	SERVICE	ACTION	TRANSLATE SOURCE	TRANSLATE DESTINA
1	-	*	*	domain udp	PASS	-	-
2	-	Trusted hosts	Gateways	FireWall1	PASS	-	-
3	-	*	*	domain tcp	PASS	-	-
4	1	zoonet	one Valid Address	http	PASS	-	one
5	1	zoonet	one Valid Address	https	PASS	-	one
6	2	zoonet	one Valid Address	ssh	PASS	-	one
7	3	one	*	all tcp	PASS	one Valid Address	-
8	4	one	*	checkpoint1/traceroute	PASS	one Valid Address	-
9	5	*	two	nntp services	PASS	-	-
10	-	*	*	icmp proto	PASS	-	-
11	6	*	*	*	DROP	-	-
12	-	*	*	*	DROP	-	-

Figure 2: The expanded rule-base, after conversion to the LFA's intermediate language.

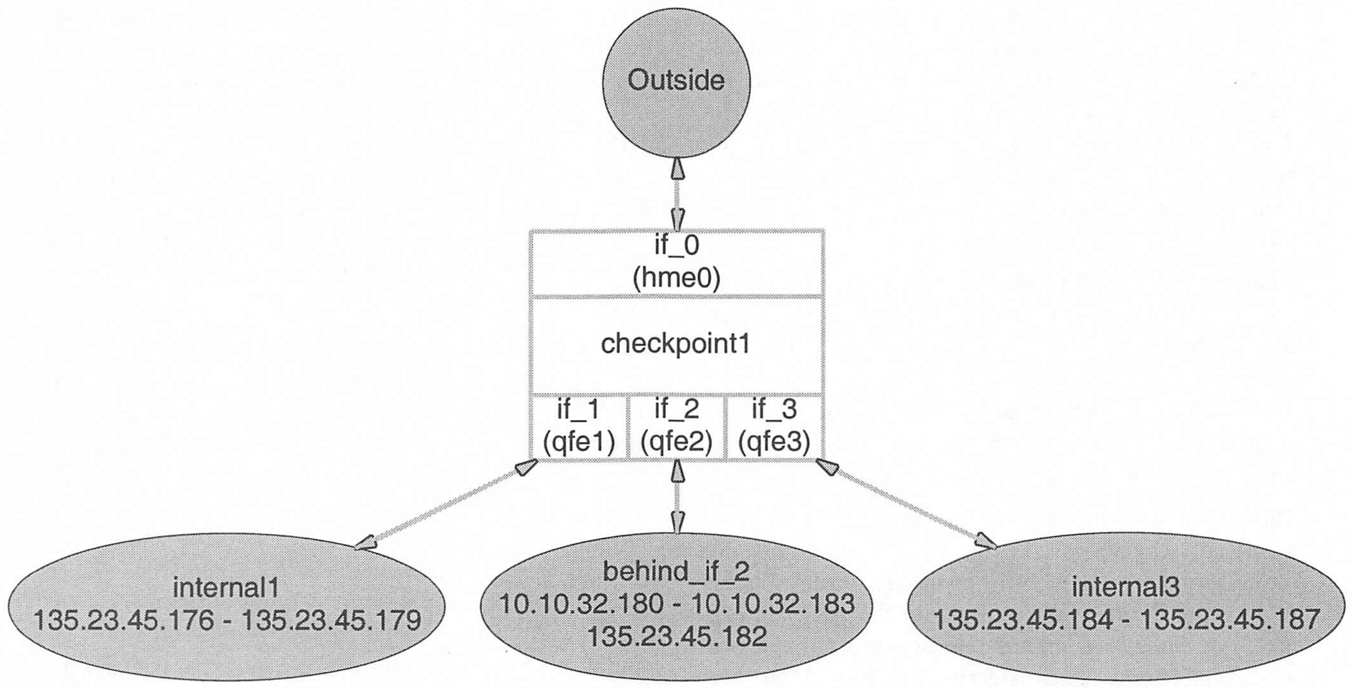
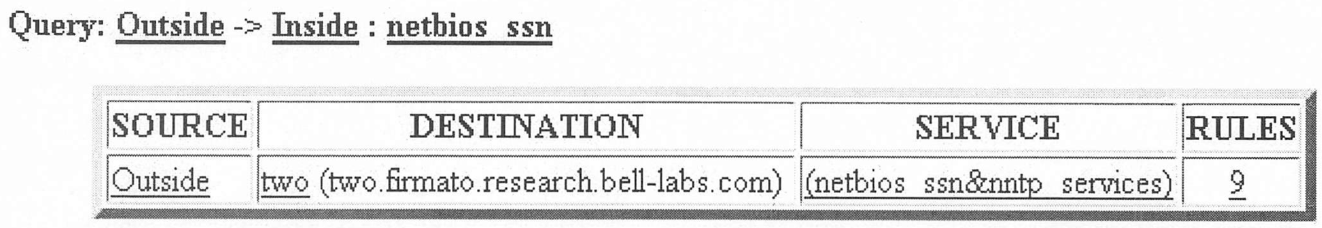
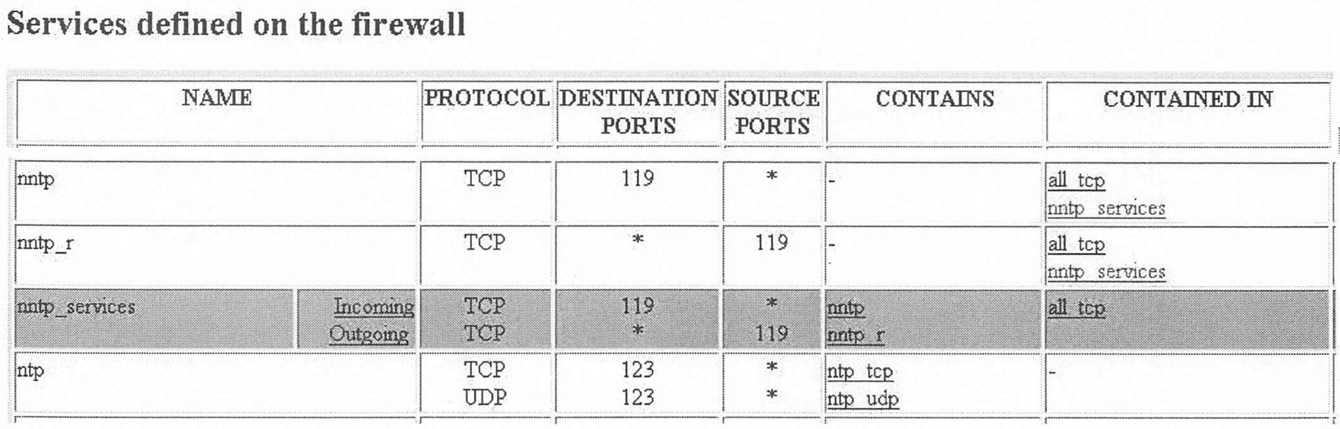


Figure 3: A diagram of the firewall's network connectivity, derived from the firewall connectivity description file.


Query: Outside -> Inside : netbios ssn

SOURCE	DESTINATION	SERVICE	RULES
Outside	two (two.firmato.research.bell-labs.com)	(netbios ssn&nntp services)	9

Figure 4: An excerpt from the "Analysis by service: Incoming" report, showing the results of the `netbios` query.


Services defined on the firewall

NAME		PROTOCOL	DESTINATION PORTS	SOURCE PORTS	CONTAINS	CONTAINED IN
nntp		TCP	119	*	-	all tcp nntp services
nntp_r		TCP	*	119	-	all tcp nntp services
nntp_services	Incoming Outgoing	TCP TCP	119 *	* 119	nntp nntp r	all tcp
ntp		TCP UDP	123 123	* *	ntp tcp ntp udp	-

Figure 5: An excerpt from the Services report, with the `nntp_services` service highlighted.

technique known as "firewalking", and is usually done using source port 53 (`dns`) which is very often open [GS98].

Remarks:

- A manual inspection of the rule-base shown in Figure 1, even by an expert auditor, is very likely to miss the vulnerability that the LFA demonstrated. The service name listed in the rule (`nntp_services`) makes sense. Even if the auditor is diligent enough to dig deeper and check the definition of the service, she would find that the port number (119) is in fact correct. It is just in the wrong column, half an inch away from being perfect.
- Similarly, a firewall probe by an active vulnerability test tool would probably also miss the vulnerability. Unlike LFA, such a tool inherently cannot test every possible combination of IP addresses and port numbers, and it would have no special reason to test the particular combination of source port 119 and destination port 139.
- We believe that the reason for the mistake in the definition of `nntp_services` is that the firewall administrator who created it was not fully aware of the implications of stateful inspection, and was probably used to configuring stateless packet filters, such as router access-control-lists. A stateful firewall (like Check Point FW-1) will automatically allow the returning packets of an open `tcp` session. A stateless access-control-list requires a separate rule for the returning packets, in which the filtering is done based on the source port (since the destination port is selected dynamically). The erroneous component of the `nntp_services` definition looks precisely like a stateless rule allowing the returning packets through the firewall.

5 Conclusions

The Lumeta Firewall Analyzer (LFA) is a novel, multi-vendor tool that simulates and analyzes the policy enforced by a firewall. The LFA takes the firewall's configuration files and routing table, parses them, and simulates the firewall's behavior against all the possible packets it could receive. The result is an explicit, cross-linked, HTML-based report showing all the types of traffic allowed in from the Internet, and all the types of traffic allowed out.

Acknowledgments

A project such as LFA is a team effort. Many people have contributed to the evolution of LFA, whether in ideas, algorithms, or code. I gratefully acknowledge the contributions of Yair Bartal, Sudip Bhattachariya, Steve Branigan, Hal Burch, Diane Burley-McGlue, Bill Cheswick, Terry Lieb, Tom Limoncelli, Ryan Martin, Alain Mayer, Kobi Nissim, Karl Siil, Bruce Wilner, and Elisha Ziskind.

References

[Bel99] S. M. Bellovin. Distributed firewalls. *;login:*, pages 39–47, November 1999.

[Bla00] BlackICE Defender. Network ICE, 2000. `http://www.networkice.com/products/blackice_defender.html/`.

[BMNW99] Y. Bartal, A. Mayer, K. Nissim, and A. Wool. *Firmato*: A novel firewall management toolkit. In *Proc. 20th IEEE Symp. on Security and Privacy*, pages 17–31, Oakland, CA, May 1999.

[Che97] Check Point FireWall-1, version 3.0. White paper, June 1997. `http://www.checkpoint.com/products/whitepapers/wp30.pdf`.

[CSS00] Cisco secure scanner 2.0, 2000. `http://www.cisco.com/warp/public/cc/pd/sqsw/nesn/index.shtml`.

[Dot01] Graphviz - open source graph drawing software. version 1.7, 2001. `http://www.research.att.com/sw/tools/graphviz/`.

[Fyo00] Fyodor. NMAP - the network mapper, 2000. `http://www.insecure.org/nmap/`.

[GKNV93] E. R. Gansner, E. Koutsofios, S. C. North, and K.-P. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, 1993.

[GS98] D. Goldsmith and M. Schiffman. Firewalking: A traceroute-like analysis of ip packet responses to determine gateway access control lists. White paper, Cambridge Technology Partners, 1998. `http://www.packetfactory.net/firewalk/`.

[IKBS00] S. Ioannidis, A. D. Keromytis, S. M. Bellovin, and J. M. Smith. Implementing a distributed firewall. In *Proc. 7th ACM Conf. Computer and Communications Security (CCS)*, Athens, Greece, November 2000.

[IOS00] Cisco IOS firewall feature set, 2000. `http://www.cisco.com/univercd/cc/td/doc/pcat/iofwfts1.htm`.

[ISS00] Internet security systems: Internet scanner, 2000. `http://documents.iss.net/literature/InternetScanner/is_ps.pdf`.

[LMF99] Lucent managed firewall, version 3.0, 1999. `http://www.lucent.com/iss/html/technical.html`.

[Lum01] Lumeta firewall analyzer, 2001. `http://www.lumeta.com/solution_firewall.html`.

[MWZ00] A. Mayer, A. Wool, and E. Ziskind. *Fang*: A firewall analysis engine. In *Proc. IEEE Symp. on Security and Privacy*, pages 177–187, Oakland, CA, May 2000.

[PIX97] Cisco's PIX firewall series and stateful firewall security. White paper, 1997. `http://www.cisco.com/warp/public/cc/pd/fw/sqfw500/tech/nat_wp.pdf`.

[RGR97] A. Rubin, D. Geer, and M. Ranum. *Web Security Sourcebook*. Wiley Computer Publishing, 1997.

[Rus00] R. Russell. Linux IPCHAINS-HOWTO, v1.0.8, July 2000. `http://www.linuxdoc.org/HOWTO/IPCHAINS-HOWTO.html`.

[XOS+00] W. Xu, S. O'Neal, J. Schoonover, S. Moser, F. Lamar, and G. Grasboeck. fwrules50, 2000. Available from `http://www.phoneboy.com/fw1/`.

[Zon00] ZoneAlarm 2.1.44. Zone Labs, 2000. `http://www.zonelabs.com/`.

Transient Addressing for Related Processes: Improved Firewalling by Using IPV6 and Multiple Addresses per Host

Peter M. Gleitz*
pmgleit@netscape.net

Steven M. Bellovin
smb@research.att.com
AT&T Labs Research

Abstract

Traditionally, hosts have tended to assign relatively few network addresses to an interface for extended periods. Encouraged by the new abundance of addressing possibilities provided by IPv6, we propose a new method, called Transient Addressing for Related Processes (TARP), whereby hosts temporarily employ and subsequently discard IPv6 addresses in servicing a client host's network requests. The method provides certain security advantages and neatly finesses some well-known firewall problems caused by dynamic port negotiation used in a variety of application protocols. A prototype implementation exists as a small set of *kame*/BSD kernel enhancements and allows socket programmers and applications nearly transparent access to TARP addressing's advantages.

1 Introduction

In the simplest of inter-networked host models, a client or server host has a single network interface with a single network address identifying the host. Even under such an elementary set-up, firewalls have traditionally faced difficulty when confronted with application protocols needing to open secondary channels. Examples abound, most notably `ftp`, but also `rsh`, RealAudio, H.323, `tftp` and the X Window System. To operate with such popular applications, firewalls have been forced either to follow the application layer protocol and configure themselves appropriately or to keep open, sometimes unnecessarily, a range of ports.

As an alternative to potentially complex, detailed, and often stateful firewall interaction, we propose a method using multiple network addresses per host to organize and simplify firewall decisions. Under our basic model, instead of trying to follow the unfolding application protocol details, the firewall makes an initial permissibility determination based on transport layer protocol and the endpoints' ports and addresses. Assuming approval of the proposed transaction, the firewall subsequently permits all traffic between the approved address pairs, irrespective of port.

*Work done while interning at AT&T Labs Research

The security concerns arising from the firewall's apparent loss of control over a session's evolving ports will be alleviated by dynamic control of the protected host's active addresses. Further, by segregating and controlling which addresses offer network services outside the firewall and which facilitate protected-host driven network requests, the architecture provides a natural address based division between potentially hostile requests from outside the bastion, and presumably benign outbound activities originating within the protected network.

To help distinguish among a protected-host's various client/server tasks, we will tie the address used by a client process to its process group identifier. This way, host addresses will come and go as part of the natural lifecycle of the processes that use them.

For example, when a TARP client starts `ftp` from behind a firewall to a similar server, also protected by a firewall, it configures a new address. The client's firewall recognizes the client's new address and records it along with the destination address. The firewall presumably grants the FTP request based on port and address criteria and subsequently passes all outbound and inbound packets between the two addresses. Further port negotiations (using the same addresses) conducted by `ftp` will have no effect on what the client side firewall passes; it will simply pass along all the server packets addressed to the client's ftp address.

The server side behaves similarly (though there are some

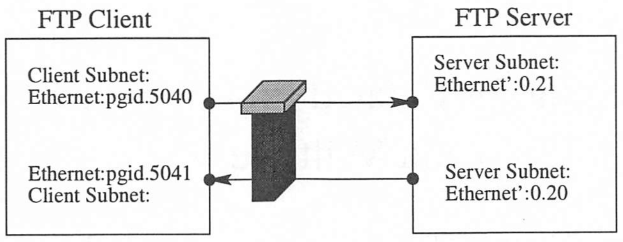


Figure 1: TARP Client FTP Through a Firewall: When the client connects to the FTP server, the client's firewall recognizes that FTP uses auxiliary ports and opens all ports between the server and the client's transient address, Client Subnet:Ethernet:pgid.The FTP thus proceeds without the firewall knowing that the client negotiated port 5041 for the data channel.

issues if different servers have different access policies); however, servers must use static addresses, largely so that clients can find them. The server's firewall sees the client address opening a connection to the server's fixed address and ftp port and decides whether to allow the connection. If the server's firewall permits the connection, it passes all subsequent packets between the client and server addresses, independent of port, including incoming calls.

Another way to view this is to adopt a virtual machine metaphor. That is, each client process group conceptually runs in a virtual machine, with an independent IP address space. This view can guide several design decisions, as shown in Figure 2.

Although IPv4 might conceivably support such a scheme, IPv6 and its large 128-bit addresses [HD98] provide a simpler opportunity to deploy TARP. Typically, the high order 64 bits of an IPv6 address provide the information needed to deliver a packet to the host's LAN, and the low order 64 bits, assigned by the site administrator, are commonly the interface's 48-bit Ethernet address padded with a constant to create a unique 64-bit interface identifier. TARP's addressing scheme exploits these pad bits to create a family of 2^{16} IPv6 addresses with the same basic uniqueness properties as Ethernet-based host identifiers. These addresses become associated with the sockets of a process group and form the foundation for TARP.

A proof-of-concept implementation exists as a small set of *kame*/BSD [KAM] kernel enhancements, and allows socket programmers and applications transparent access to TARP's advantages. The basic server/client programs supplied with the *kame* IPv6 package (`tftp`, `ftp`, `rlogin`, `rsh`, `telnet`, `sendmail`, `ssh`, `mozilla`, `inetd`) all work with the modified kernel, and none of them required modification to run.

2 Multiple Addresses per Host and a Large Family of IPv6 Addresses

The notion of assigning multiple IP addresses to a (non-routing) host is not new. In 1994, RFC 1681 [Bel94] described a number of potential uses and benefits of assigning multiple IP addresses to a host, and Stevens' standard socket programming text [Ste90] is peppered with some of the subtleties of socket programming on multihomed servers. The IPv6 addressing model specifically supports assignment of "multiple IPv6 addresses of any type (unicast, anycast, and multicast) or scope" [HD98] to a single interface. Although primarily intended to facilitate renumbering, this IPv6 feature can also be used to simplify firewall interaction via TARP.

To form addresses based on unique interface identifiers, RFC 2373 [HD98] recommends using, whenever possible, the 48-bit IEEE 802/Ethernet MAC addresses, and the RFC offers the IEEE EUI-64 [Ins97] as one way to create a globally unique 64-bit identifier from the Ethernet address. The suggested method consists of inserting the two-byte padding pattern $FFFE_{16}$ after the third byte of the the Ethernet address.

To create TARP addresses, we actively vary the pad bytes to select from a large family of IPv6 addresses unique to an interface. We also use a slightly different ordering than the examples in RFC 2373. The first six bytes of our addresses are simply the interface's Ethernet address with the last two bytes varying as needed. Re-ordering the bits that vary to the least significant bits has the benefit of allowing the last hop router to use a single table entry with a 112-bit prefix to send all such packets to a single host.

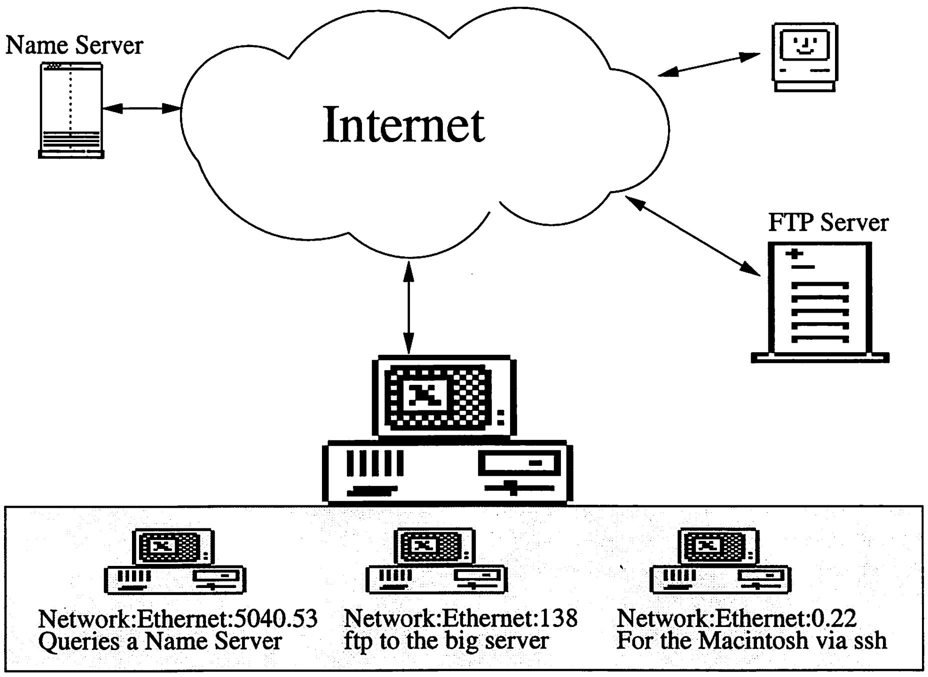


Figure 2: An Example Under the Virtual Machine Model:Viewing each process group as a virtual machine, the figure shows a single TARP host involved in three distinct network sessions run by separate process groups 1) an FTP session from the TARP client to a server across the WAN ; 2) the TARP client serving an ssh session to a Macintosh client; 3) A DNS request originating from a process within process group 5040.

3 Some Notation

We now introduce some notation. Suppose a host interface has an Ethernet address of 00:10:4b:63:80:4b, a 64-bit network prefix length and a subnet prefix of 1111:2:3:4. Then the TARP address family available for this interface ranges from 1111:2:3:4:10:4b63:804b:0 through 1111:2:3:4:10:4b63:804b:ffff. We describe the general concept of an address like 1111:2:3:4:10:4b63:804b:0, as *Subnet Prefix:Ethernet Address:0* and when the least significant bits of an address correspond to the process group identifier using the address, we write *Subnet Prefix:Ethernet Address:pgid*. When the discussion requires some notion of a port, a dot after the final address bytes delimits the port, as in *Subnet Prefix:Ethernet Address:0.21* for port 21, or *Subnet Prefix:Ethernet Address:0.** meaning any port on the address. As a further notational imprecision, quantities like *Subnet Prefix* and *Ethernet Address* can be enhanced to refer to some particular network, interface, or host context, as in *Subnet Prefix Network 1*, or *Ethernet Address Host 2, etc.*

4 Using a Large Family of Host Addresses

Host usage of this this large address family depends on whether the host process acts as client or server. In general, client applications select actively from the pool of available addresses and dynamically configure the appropriate interface. Server processes will use a designated static address.

Our decision to connect the address used by a client process to its process group was driven only by a desire to have a group of related processes use the same address. Process groups have no particular special properties, other than being somehow "related" (on Unix systems, they represent the processes descended from a single command line), and the basic concept of transient addressing extends to other similar, natural process relationships. As the basis for the UNIX process group indexing scheme, processes could, in principle, have their own IP addresses, although the clutter and possible performance penalties of establishing a per-process address seem excessive. In a multi-user environment, UIDs, credentials, or session identifiers could sensibly anchor similar addressing schemes, as could a new (as yet unspecified and only dimly imagined) set of kernel concepts and data structure for organizing related processes.

More fundmanentally, the requirement is that the addresses assigned to "related programs"—ones that somehow "expect" to be at the same place—be stable, but that "different" programs have different addresses. Thus, a Web browser structured as multiple processes should have a single IP address, while individual `ftp` commands should have different addresses.

Address organization by process group has the advantage that the kernel already supports the notion that a collection of processes is somehow related. The data structures are stable in the UNIX kernel, and bolting the addressing scheme onto them posed little threat to the processing flow for establishment, maintenance, and appropriate removal of the fundamental data structures managing process groups. We are unaware of any basic conflicts created by attaching our concepts to process groups, and found them lively and flexible enough to stress the kernel enhancements dealing with address management.

4.1 How a Client Uses the Large Family of Addresses

When a client implicitly binds a local address to a socket, the kernel decides which interface to use. Our modified kernel determines what process group the binding client process belongs to, and uses the corresponding *Subnet Prefix:Ethernet address:pgid* address for the socket. For a process group's first such socket, this involves assigning a new IPv6 address to the appropriate interface and joining the corresponding Solicited-Node multicast address required by the IPv6 specification [HD98]. Note that by only assigning a TARP address in the case of an implicit bind, the kernel still allows explicit binding to any valid host address a socket program can identify. With an address established for the process group, any other implicitly bound sockets arising from the same process group will use the same address. The address persists on the interface until the process group leader terminates, when the address and its Solicited-Node multicast address are removed.

4.2 How a Server Uses the Large Family of Addresses

Server interactions passing through the firewall must occur on the reserved *Subnet Prefix:Ethernet Address:0* address. This requirement serves several useful purposes. First, connecting to a server with a dynamic address is like hitting a moving target, so fixing the address allows clients to find servers. Next, fixing the server address (both listening and responding address) further assists the expected server/firewall interaction by omitting any need for server applications to inform interested firewalls of their addressing dynamics. Finally, it provides control for firewalled hosts running servers using wildcard listening sockets.

The control requirements are the most subtle, and an example of what could be dubbed "coat tail riding" will clarify the importance of restricting external server activity to a fixed address. Consider an installation with a policy freely allowing outgoing `ftp`, but not incoming `telnet` from outside its firewall. A host inside the installation opens an FTP connection from *Protected Installation Prefix:Ethernet Address:pgid.5040* to a hacker-compromised site, *Compromised FTP Server.21*. This complies with the installation's security policy, and the firewall enables the rule allowing all traffic between the protected host and the compromised server, {*Protected Installation Prefix:Ethernet Address:pgid.* * $\longleftrightarrow$ *Compromised FTP Server.**}. A telnet server listening to a wildcard socket on the `ftp` client machine could then receive and serve connection requests from the compromised FTP server to *Protected Installation Prefix:Ethernet Address:pgid.23* in violation of the installation's telnet policy. We prevent a server from riding back on the client's coat tails by forcing server activities to occur on the :0 address and preventing a server's wild card listening sockets from connecting to inappropriate TARP addresses. Section 5.4 describes a set of socket matching rules that prevent coat tail riding. Note we cannot simply forbid connections to *Protected Installation Prefix:Ethernet Address:pgid* as active mode FTP will require that the compromised server connect back to the client.

The address based firewall rule and designated server address principles lack fine discrimination concerning what services will be served outside the protected bastion. Once a client opens a connection through the firewall to any service, the firewall rule subsequently allows the same client through to any port. In another example, suppose the protected host's security policy allows `ftp` connections from the outside, but not `telnet`. Any authorized user could start an ftp from the outside to *Protected Installation Prefix:Ethernet Address:0.21*, causing the firewall to allow all traffic between the ftp client and the protected server. With all ports now open to the FTP client, it can now start a `telnet` session from the outside, unimpeded by the firewall. This problem of finer discrimination of service access has some reasonable remedies short of following application-level protocols, and we discuss them in the security section below.

5 TARP and Sockets

5.1 Overview

Our TARP implementation rests on a set of kernel enhancements at the socket level and below. It relies extensively on the socket data structure's process group identifier, and every AF_INET6 socket receives the process group identifier of the process group creating it. The main intent has been for socket programs to receive, without further programmer intervention, the appropriate TARP address when simply using the vanilla Socket API.[1]

As illustrated above, it matters greatly to the addressing scheme's security concept whether a process acts as a server or client. As a result the kernel tries to classify a process group as either a client process group or a server process group, and typically assigns a classification to a process group according to the first SOCK_STREAM or SOCK_DGRAM socket usage of a process in a process group. A process group typically exists in an indeterminate state until one of its processes first reads or writes a socket. The exception to the usual indeterminacy occurs when a server spawns a new process group. In this case, the new process group inherits its server state from the parent process group. If a process group's first datagram or stream socket usage is a read, the enhanced kernel classifies that process group as a server process group. Similarly, if the first such socket usage writes a socket, then the process group receives a client classification. The implementation offers system calls to manipulate a process group's client server state for socket applications ill-served by the default classification rules, but experience has so far shown them needless: except for `auth` (see Section 6.1), all tested applications worked without change.

[1] In this prototype, we have not fully checked if SS_ASYNC's usage of the sockets process group components may conflict.

5.2 Client Sockets

Client applications wishing to use a TARP address simply create a socket and first write to it (using `write`, `writev`, `sendmsg`, or `sendto`) or call `connect` so that the socket receives an implicit `bind`. During the process of implicitly binding the socket to a local address, the kernel assigns and configures the appropriate TARP address.

5.3 Server Sockets

Like client sockets, server sockets can receive the source address of their outgoing packets via an implicit bind, but sometimes, this produces undesirable results. Under the general model that servers will create sockets somehow listening on some limited set of addresses for contact from clients, one of our goals is to insure that servers will listen to and respond from the same reserved server address that was contacted by the client. Achieving this goal depends on the transport layer protocol, the Unix flavor, and the server's socket programming style.

TCP servers pose little problem as the connection framework provides adequate structure to determine the correct address for a listening socket to bind to (presumably a reserved server address contacted by a client). We do not force the `accept` to occur on a server address, but rather view that decision as a administrative choice for the firewall. By allowing connections to valid addresses outside the TARP address family but sharing the same interface, we can also support multi-homed TCP servers. We expect (but do not force) that installations using these extra addresses will have firewalls routinely blocking all traffic involving these non-process group based addresses so that they will be served only from within the protected enclave.

UDP client/server applications are slightly more problematic, mostly because the BSD derived kernels do nothing to force a multihomed UDP server to respond from the same address as the destination address of the prompting packet[Ste90, p. 220]. Coupled with the reality that many common UDP servers let the kernel choose the server's responding address, this effectively causes problems for clients attempting to use UDP sockets to communicate with multi-homed UDP servers running a kernel supporting TARP addressing. The BSD based kernels choose a reply address from the outgoing interface which does not necessarily mean replying with either the same address as the client's destination address or the designated :0 server address. If a client contacts a UDP server on *Subnet Prefix:Ethernet Address:0*, the kernel must insure that the response returns from the same server address or the firewall rule set will likely block the response.

For UDP servers, the classification of a process group as a server process forces response on the server address, achieving the desired result. The whole classification scheme is mostly forced by BSD's carefree attitude about ensuring that the UDP server responds from the same address contacted by the client.

5.4 Matching Protocol Control Blocks for Inbound Data to Sockets

To read incoming data, a process typically creates a socket and places it in a state (often blocking, but perhaps polling) awaiting network data. The kernel monitors incoming IP data, classifies it according to transport layer protocol and ports, and finally matches it to a list of available sockets. To work well with TARP addressing, the standard BSD rules for matching network data to sockets need refinement, and the new assignment rules are described below.

These assignment rules enforce several operating principles. First, they must prevent coat tail riding by providing the control necessary to prevent wildcard listening sockets from providing network services on a process group based address. Second, the rules must also permit connections to servers on the *Ethernet:0* address. Third, they need to connect the appropriate incoming packets to sockets listening specifically to a TARP address. Finally (and optionally), because servers may wish to provide separate services inside the protected bastion, the rules need to allow a server to match sockets listening to a valid address outside the TARP address family. The rule set below achieves these goals.

1. A socket with a socket pair ({*source address.source port, destination address.destination port*}) exactly matching a datagram's source and destination ports and addresses receives the datagram payload. This rule typically applies to TCP sockets from established connections, for example.
2. Bound Listening Sockets: If no match is found in Rule 1, the sockets listening to a particular address are examined sequentially for the first match against the following ordered rule set:
   a. A socket listening to {*.*, *Subnet Prefix:Ethernet Address:0.port*} receives the data sent by any client to the server on that port. This rule applies to server connection establishment, and UDP servers on the server address.
   b. A socket belonging to process group *pgid*, and listening to {*.*, *Subnet Prefix:Ethernet Address:pgid.port*} gets data sent to that port and address. This rule allows TARP address clients to use listening sockets to accept connections from servers, like an `ftp` client might do.
   c. A datagram addressed to a socket listening to the specified port on a valid host address outside the TARP address family will receive data sent to that port and address. This is intended to handle requirements to provide services not available outside the protected bastion to internal hosts.
3. Wildcard Listening Sockets: If no match for any of the sockets is found in Rule 2, then a socket listening to {*.*, *.*port*} can match if and only if:
   a. The datagram from the client is addressed to the server address.
   b. The datagram is addressed to a valid TARP address and the socket belongs to the process group designated by the address.
   c. The datagram destination address is outside the TARP Family but a valid host address.

   Rule 3 is a catch-all and the last socket matching under Rule 3(a), 3(b), or 3(c) receives the datagram.

An example shows how these rules effectively prohibit coat tail riding. Reconsider the example of Section 4.2, where a protected client runs both telnet and FTP servers on a wildcard address, but allows only FTP to outside the firewall. After the protected client starts an `ftp` session to the compromised FTP server, the compromised site attempts to open a connection to *Protected Installation Prefix:Ethernet Address:pgid.23*, and the firewall passes the malicious SYN packet according to the rule legitimately established by the `ftp` allowing {*Protected Installation Prefix:Ethernet Address:pgid.** $\longleftrightarrow$ *Compromised FTP Server.**}. As the `telnet` server is listening with a wildcard socket, the the first two rules (each requiring a specified destination address) will fail to match the SYN packet. The only remaining candidate is Rule 3(b), but this fails to match because the telnet server's listening socket will belong to a different process group than the ftp. Thus the packet intending to ride the coat tails of the open firewall filter is never delivered to the telnet server socket and the connection is never opened.

Rules2(c) and 3(c) are optional and permit a host to offer services on an address outside the TARP family, presumably to machines within the protected enclave. They should be filtered outside the protected enclave and also admit the possibility of firewall misconfiguration. For installations not requiring their functionality, a compile option can remove them and protect against misconfiguration dangers.

5.5 Aids for Socket Programmers

Although the enhanced kernel modifications work sensibly with a variety of unmodified IPv6 server/client software, socket programmers using TARP addresses may need to be mindful of certain subtleties. Socket programmers writing applications that fork new processes and subsequently assign the forked processes to new process groups will will need to remember that the resulting sockets may not share the same IP address (if that matters to them or their application). Additionally, the inheritance of a parent's process group server state could conceivably create problems if a server forks a process that needs to act as a client.

As a convenience and measure of control, the modified kernel provides a system call allowing a program to request either server or client status. So far, the client/server applications tested have required no such manipulations to use the correct addresses, suggesting that the kernel rules generally provide the desired results. During development, the modified kernel also contained a setsockopt call allowing programmer control of whether a socket uses the server addresss, but this control appeared superfluous in light of the server inheritance rule and was removed pending demonstration of its utility.

Host applications may also explicitly bind to any address they could otherwise bind to with an unmodified kernel, although doing so will likely produce useless results. For example, an application could bind to the address used by a separate process group, but the resulting socket would (absent process group manipulation) have an address with the process group component not equal to the socket's process group. This would always fail the protocol control block matching algorithm joining sockets and protocol control blocks, and so no data could flow to the socket. (A simple code change could prohibit the ability to misbind, to avoid violating the rule of least surprise.)

6 Experience with Client/Server Interactions

6.1 TCP Applications

Our experience is primarily limited to some of the IPv6 server/client inetd programs available from *kame* : inetd, tftp, ftp, rlogin, rsh, and telnet, and auth, although we also worked successfully with IPv6 ports of ssh and sendmail as well as with a web browser mozilla and a web proxy wwwoffled. Except for auth[Joh93], the TCP applications all appear to run reasonably well, unmodified, with the enhanced TARP address kernel. The UDP client/server pairs also ported well, though with some mild restrictions concerning the server's responding address.

The problems encountered by the unmodified applications depend variously upon the session layer protocol, the application layer protocol (auth), and whether the application uses address based authentication (*e.g.* rsh and rlogin). Cognizant of the terrible security properties of address based authentication [Bel89], we question the wisdom of bothering to fix rsh and rlogin, especially with ssh working, but their functionality can be restored with modest DNS adaptations discussed in Section 7.3. We have not performed exhaustive testing. (We do note that ssh [Ylo96] can use cryptogrpahically-protected name based authentication; similarly, rlogin et al. are secure when protected by IPsec [KA98]. If use of such protocols is considered desirable, we may need to reopen this issue.)

Of the TCP applications, ssh, ftp, and telnet, sendmail, and ssh appear to function normally with the modified kernel, while rsh and rlogin both suffer mildly from authentication problems already described.

The auth server presents insurmountable problems, mainly because its poor interaction with TARP addressing. Consider a server installation wishing to fire an Ident query to a TARP client's auth server about a connection coming from, say, *Subnet Prefix:Ethernet Address:pgid* . Ident queries consist only of the port numbers; the query addresses are implied by the source and destination address of the queries. The querying host has two choices for an address to query, neither of which work. Querying the *Subnet Prefix:Ethernet Address:pgid* will fail, as there will be no server on the process group address. Querying the server address, *Subnet Prefix:Ethernet Address:0* yields a null result, as the connection does not originate from that address. This failure seems a small loss, as the quality of the information returned from the auth server has always been greatly disparaged (c.f.[CB94, GS96], and even its defining RFC [Joh93]). As a workaround, client installations feeling an obligation to reply to Ident queries should have little trouble hacking a small Ident server into client applications, so that the application itself replies to Ident queries on its process group address. Of course, such an implementation would require a setuid helper program.

6.2 UDP Applications

By restricting UDP clients and servers to asking only for services and responding to requests (respectively) from an interface's designated server address, then all the *kame*/BSD UDP client/server applications work correctly, unmodified. We wrote simple socket clients to communicate with inetd's built-in servers, and (except for auth) succeeded with both the TCP and UDP versions of: time, echo, daytime, and chargen.

We should distinguish that our UDP clients used *unconnected* UDP sockets when communicating with the built-in servers. When requesting service from a valid address outside the TARP Family, the clients executed a sendto followed by a read. Following the rule that processes first reading then writing sockets are servers, the server responded from its *Subnet Prefix:Ethernet Address:0* address, which the client successfully read from its unconnected socket.

Had we instead used a client socket connected to the valid non-TARP address where the UDP service request was sent, the UDP client server transaction would have failed when the reply returned from the server address. This problem with connected sockets is not isolated to the modified kernel, but also exists when interacting with multihomed Berkeley-derived UDP servers [Ste90, pp. 219–220]. Under BSD, the server is only guaranteed to reply from an address chosen from the same interface as the contacted server address. The server can reply with an address other than the one contacted by the client (but on the same interface) and a connected UDP socket will fail to read the server's reply.

6.3 ICMPv6

Some network services are bound to the existence of an IP address, rather than to a specific port number. For example, ICMP messages are received by a host, rather than by a specific process. How should messages sent to TARP addresses be treated?

Our primary guiding principle is that of the virtual machine, though for implementation reasons we cannot always follow this. Thus, a "ping" message sent to a TARP address should be replied to, because a real machine with that address would respond.

Some ICMP messages create state on the receiving host. For example, ICMP Redirect changes the local routing table. Given the security risks posed by this message [Bel89], it would be nice if these changes could be restricted to the particular process group to which the messages were addressed, but that would require substantial kernel changes. Furthermore, there are distinct advantages to making certain state information global, such as that distributed by Path MTU [MDM96]. Further work is needed in this area.

7 Network Interaction

7.1 Firewall Interaction

TARP was designed for firewalls, with the goal of providing a mechanism for firewalls to make sensible security decisions without following application layer protocols. The resulting firewall interactions are intended to be straightforward and simple. The main precept is to use address based filtering after an initial authorization. For a connection based protocol like TCP, this means port information need only be examined at connection establishment.

For outbound data, the firewall only needs to examine the outbound destination port, determine whether the originating address belongs to a family of TARP addresses, and know whether the protocol involved uses auxiliary data connections involving other ports. Assuming it approves the protected client's transaction and the underlying protocol requires dynamic port negotiation, the firewall simply permits any incoming traffic to that address, regardless of port. The firewall no longer needs to know any protocol details; it simply needs to know that the protocol involves secondary channels. For services using only fixed ports, the firewall can filter traditionally.

For service requests originating from outside the protected bastion, the firewall typically would reject all requests for the services of an address other than a designated server address. For authorized services, the firewall permits packets to flow freely between a server address and the client outside. For example, an FTP server host administratively configured to provide no other services outside the firewall would reject all inbound connection attempts and UDP packets other than those to the FTP server port. Following a successful connection, outbound packets to the client from *Subnet Prefix:Ethernet FTP Server Host:0.** are freely allowed between the approved server and client.

The firewall can use several approaches to revoke authorization. TCP connections are easiest: If the packet triggering authorization comes from a TCP connection, the firewall simply disallows (pending possible future authorization) packets between the two addresses when all related connections (as determined by address) terminate. Additionally, and more appropriate for UDP and ICMP, a simple timer mechanism can revoke authorization some number of minutes after the last use of an address. Finally, a protected host can explicitly release an address, upon process group termination.

Having a host communicate its release of an address provides the firewall the best general understanding of when to terminate authorization, but it requires the protected host to know how to reach its firewall(s). The best control exists in situations where all relevant firewalls are either embedded in the host (like a Distributed Firewall [Bel99]) or or share the same link layer and are able to see the link local ICMPv6 broadcast as a result of the host leaving the Solicited-Node multicast address. Here, the firewall either knows or has an inkling of the address being removed at process group termination. The embedded firewall can simply de-authorize the canceled address from within the kernel, while the link local firewall will need to match the Multicast Listener Done message to its authorized addresses, and cancel upon determining their unreachability.

For installations where routers lie between the protected host and the relevant firewall, hosts wishing to communicate address revocation will need more complex interaction mechanism with their firewalls.

Using a traditional proxy firewall to protect hosts with TARP addresses works as before, but will require some flexibility in configuration to accommodate the large number of addresses that will be coming and going. For example, certain outbound filter rules will need to apply for *Subnet Prefix:Ethernet:** where before they perhaps pertained only to a single address per host.

Note well that clients are usually well protected, even if the firewall is slow revoking authorization. This is because when a process group terminates, its corresponding address is scrubbed from the client's interface, so even if the firewall is tardy blocking the inbound packets, they fall on deaf ears when reaching the host. Any nefarious or spurious inbound packets to unconfigured addresses should ultimately die an unconsumed death. *This is the fundamental reason why TARP-based hosts can simplify firewalls.*

Because we use only the initial address and port of a connection, we have prevented new application protocols from obsoleting otherwise adequate firewall software. Instead of requiring complex (and possibly buggy and insecure) new application protocol-aware software to be written and installed in the firewall, to keep up with new application protocol developments, the firewall administrator need only know what port the service uses to start its transactions and filter accordingly. Thus, only minor firewall configuration changes will accommodate a large class of new, unforseeable applications. (This work does not discuss application-specific audit trails or intrusion detection system events that an application-aware firewall may collect today.)

7.2 Router Interaction

At one level, routers are not affected by TARP addresses. They will see an IP address for which they have no corresponding link-layer address cached; accordingly, they will resort to the Neighbor Discovery Protocol [NNS98], in normal fashion. But TARP-aware routers and protocols can do better.

Widespread use of TARP would require considerably more storage on routers for link-layer addresses. But this set of IP addresses all share a common prefix and a common link-layer address. Accordingly, routers could employ a single mapping of *prefix* to link-layer address. Unfortunately, that is not supported by NDP. If necessary in a given environment, this could be faked by having a host pretend to be a stub router; however, this would require the host to participate in routing protocols, which is generally considered to be a bad idea. A better solution would be to extend NDP to handle host address prefix lengths.

7.3 DNS Interoperation

As evident in the discussion concerning `rlogin` and `rsh`, TARP Addressing encounters problems interacting with DNS. Of particular concern is the dubious practice of so-called "double-reverse lookups" (cf. [ZCC00] and [GS96]) where a DNS client attempts address based authentication by first looking up the hostname for an IP address and subsequently checking the IP address(es) for that hostname, requiring a match to proceed. But for double-reverse lookups, most dynamic address clients would probably work happily by simply registering their server address with the appropriate name server; a wildcard PTR record would yield the same name for all possible TARP addresses.

IPv6-to-hostname lookups pose no problem, as the DNS specification for IPv6 [TH95] provides the necessary wildcarding. For example a name server could associate a wildcard to a PTR record for, say, **.12-nibble Ethernet Address.16-nibble Subnet Prefix.IP6.ARPA* (ordered appropriately) and correctly return the hostname for a host's Process Group Dynamic Addresses. Another possibility is to query the host directly with an ICMPv6 FQDN query, as described in an IETF draft "IPv6 Node Information Queries" [Cra99] and already implemented by *kame*.

Hostname-to-address queries are more problematic. A standalone name server presumably has no information about a host's active addresses, and absent refinements in the name server/resolver software and protocols, a complete reply to a hostname-to-address query would consist of all 2^{16} Process Group Dynamic Addresses. Unfortunately, simply listing a host's server address is insufficient, as a double-reverse lookup will find inconsistency when comparing a client's dynamic address to the its server address.

Three straightforward solutions to the basic problem come to mind. The simplest treats a Process Group Dynamic Address client as a domain. Its parent DNS server delegates authority to the client and simply refers queries to the clients themselves for final resolution. The clients can then run a pared down name server which responds with the appropriate set of addresses for a name-to-address query.

In another method, the TARP clients use the DNS UPDATE message [VET+97] to inform the relevant name servers of their evolving address state. It is not clear how well this would work in practice, as problems could arise in situations where the querying agent has a faster path to the name server than the TARP client. A clever TARP client could conceivably delay outgoing datagrams from a new TARP address until a DNS update could be verified, but this might prove unnecessarily cumbersome.

A third method embeds the process group portion of the address into a synthetic hostname and places the corresponding 2^{16} possible hostnames as entries in the nameserver database. For example the fictitious TARP client host *foo.bar.com* would have name server entries for *0%foo.bar.com - 65535%foo.bar.com*, where we use the % to mean a meta-character (determined locally by bar.com) denoting that the hostname uses Process Group Dynamic Addressing. The name server would conceptually have 2^{16} PTR records mapping each Dynamic Address family member to its corresponding expanded hostname (e.g. pgid%foo.bar.com) The cross-check query invokes an address-to-hostname query for *foo.bar.com Subnet Prefix:foo.bar.com Ethernet Address:pgid*, which eventually returns *pgid%foo.bar.com*; this is followed by a name-to-address query about *pgid%foo.bar.com*, which will provide a consistent check if the name servers are configured correctly.

This third method scales poorly, taxing the DNS servers serving TARP client domains by forcing them to store 2^{17} records per client. While some of this can be dealt with by "syntactic sugar"—the servers need not store all 2^{17} records, but can simply generate them on the fly—we are left without a canonical name for the host. This affects host configuration files (`/etc/hosts`, `.rhosts`, *etc.*), email, and other services that require one true name for each machine. (In a sense, this is carrying our virtual machine metaphor too far.)

Finally, we note that the problem only arises because hosts use name-based authentication, and hence need the extra protection of the double-reverse lookup. If this practice were to be abandoned—and we strongly suggest that that be done in any event—the name-to-address lookup could be omitted, thereby eliminating the problem.

8 Implementation Details and Performance

8.1 Implementation

Our testbed implementation consists of a handful of subroutines and about 400 lines of C code inside the *kame*/FreeBSD 3.4 kernel. The kernel enhancements appear to port easily to the other BSD families *kame* supports (NetBSD, BSD/OS and OpenBSD). The essential code flow remains unchanged, and the bulk of the software changes sit inside a single *kame* subroutine, `in6_selectsrc` (basically an IPv6 re-work of parts of the function of `in_pcbconnect` [WS95]), called when assigning a local IPv6 address to a protocol control block.

The *kame*/FreeBSD code has a parameter for the maximum process identifier, PID_MAX, which comes configured at 99999. Our implementation reduces this maximum to 65535 to insure the resulting process group identifiers fit entirely into the corresponding two bytes of the TARP address.

Given the uniqueness properties implicit the Ethernet address based addresses, the implementation does not perform Duplicate Address Detection [TN98] when adding a TARP address to an interface. This enhances performance by preventing unnecessary delays when configuring an interface with a new address, and yet still complies with the RFC's notion that addresses based on unique interface identifiers need only check for duplicate addresses at initialization ([TN98], Section 5.4). When bringing up an interface, the *kame* IPv6 implementation configures it with a proper Ethernet-derived, EUI-64 compliant link local address. This address receives proper duplicate address detection, and so any (presumably relatively rare) problems caused by locally duplicate Ethernet addresses should be caught when the interface goes up.

Regrettably, the IPv6 Address Autoconfiguration Standard [TN98] lacks a sub-net mask capability that would allow hosts to reserve a range of addresses under a mask. Were such a facility available, a TARP host could make a single duplicate address detection query to reserve a whole range of an address family with a single query. In this case, hosts would be freed of needing to use the Ethernet address-based connection for its implied uniqueness properties, and could still safely reserve a range of without fear of address collisions. Local administrators would then have the freedom to assign the host address portion according to their own rules, and could conceivably use even larger address families.

We could, in principle, do duplicate address detection for each TARP address. However, the overhead, and particularly the 1-second timeout, are prohibitive.

There is some conflict between this scheme and the IPv6 privacy extension standard [ND01]. The easiest solution is to use the suggested algorithms to generate just a replacement for the MAC address portion of the address, rather than the full low-order 64 bits.

8.2 Performance

In our tests thus far, there has been virtually no impact on performance. It is clear, however, that current kernel algorithms will not scale well. The list of IP addresses per interface is kept on a linked list, which implies a linear search for each packet received. Clearly, this is inadequate if there are many process groups active at any time. The obvious alternative is to use a more sophisticated data structure, though it would have to be one that permitted speedy additions and deletions.

An alternative would be to compare the prefix of the address in incoming packets to the base address of the interface, and use existing speedy look-up mechanisms to ascertain if the associated process group exists. That is, assume that a packet is destined for a machine if the base address is valid and the process group exists, rather than checking if such a process group has actually performed network operations.

There is one other area of some concern. As noted earlier, for each new TARP address allocated it is necessary to join the appropriate Solicited-Node multicast address group. For some hardware designs, it is necessary to load the group address onto the controller chip. Depending on the chip and driver design, this may be an expensive operation.

9 IPSEC Interaction

Successful IPSEC interaction essentially depends on the ability of hosts running TARP addressing to conduct the necessary key exchanges. Fortunately, ISAKMP and IKE provide an adequate framework to support TARP. A simple solution uses either of ISAKMP's [Pip98] address range or subnet identification payloads to designate that the ISAKMP peer negotiating the key exchange is the address range or subnet of the TARP Family. Using keys negotiated for the address family thus permits processes to use IPSEC in conjunction with TARP addresses.

Depending on the threat model, installations may wish to eschew IKE's Base Quick Mode [HC98, p. 16]. Lacking perfect forward secrecy, Base Quick Mode admits the possibility that an authorized host process able to obtain its keying material can use that knowledge to determine keying material for other processes, including those belonging to other users. The Quick Mode key exchange payload option [HC98] prevents this problem by providing the necessary perfect forward secrecy at the cost of an additional exponentiation. Alternatively, use of cryptographically strong random number generators, ciphers resistant to chosen-plaintext attacks, and suitable crypto-APIs (i.e., those that will not, under any circumstances, disclose a session key to an application) can prevent this attack.

10 Interaction with Other Systems

At this point in the evolution of IPv6, it would not be acceptable to introduce a new scheme that would break compatibility with existing systems. For the most part, we have not done so.

Servers do not notice anything different, with the possible exception of contacts from many more clients. That depends on whether the server is noting addresses or names, and in the latter case, on what mechanism is selected for address-to-name resolution.

Servers that do rely on host names for authentication may have problems. As noted, we recommend that that practice be abandoned in any event; we do not consider its difficulties to be a disadvantage of our scheme.

11 Security Implications

In some sense, the simplicity of the security enabled depends on the assortment and type of network services a protected host needs to provide. TARP addressing offers simpler, more robust protection for hosts acting primarily as clients rather than servers. This does not mean that many server configurations cannot receive adequate protection using the method and a compatible firewall, rather that servers will need to be much more careful about their configuration to achieve results comparable to hosts acting entirely as clients.

11.1 Client Security

Clients offering no network services can be well-protected by TARP addressing and an accompanying firewall. Such clients will also operate reasonably unimpeded by their firewalls. The main observation is that absent user-directed activities, the typical client providing no network services (a dedicated workstation, for example) could be protected from all TCP and UDP datagrams from outside the protected bastion. It thus shares many of the security features of the proverbial unconnected host. Yet if some user directed activity makes it necessary to leave the protected enclave (a user wishes to run `ftp` to retrieve a file from the Internet for example), sufficient connectivity is enabled to allow the user to conduct transactions, unimpeded by tight security policies concerning connectivity to the outside. Furthermore any vulnerability created by the FTP client's network activity terminates with the FTP client process.

11.2 Server Security

For services using an evolving set of ports, the principal filtering domain has shifted from ports to addresses. Many servers are programmed with wildcard listening sockets, and the ensuing address promiscuity of these sockets poses severe challenges for address based filtering. The problem with protecting servers is that after an initial connection, the server's firewall no longer follows the ports of the packets coming and going to the server. If an initial connection is permitted from a host address to a server, then all subsequent packets between the two addresses will flow freely. This can result in inadequate control, as shown in the following example.

Again, consider a server that wishes to permit incoming access to `ftpd` services from the network, but not `telnetd`. A devious client could start an FTP session to the server, which the server firewall permits, and once the firewall enables the rule {Devious Host.* $\longleftrightarrow$ Server:0.*} the client can now connect to the server's `telnet` port from the network in violation of the intended security policy.

One obvious solution to this problem is to use an embedded server access control mechanism (something like TCP Wrappers [Ven92]) to regulate what will be served to the outside on the *Ethernet:0* address. After breezing by the firewall with an attempted `telnet` request, the previous paragraph's devious client could be refused a connection to `telnetd` according to its source address. Similarly, the `inetd` super-server has a address option that will bind the server to a particular address, and this should provide adequate control for services running under `inetd`.

A simpler approach would be for the firewall to continue to monitor connection attempts to well-known services and filter accordingly. This deviates from the concept of filtering solely based on address, but not necessarily to the point of continuously following application level protocols. In our example, connection attempts to all privileged ports other than the ftp ports would be blocked at the firewall, and the ftp could proceed.

Clearly, relying solely on a packet filtering approach to provide server security becomes more unworkable as the variety of externally-accessible services grows. As the number of services increases, sensible packet filter rules become more difficult to specify, and the likelihood of consistency problems caused by interaction effects increases correspondingly. This is an old and familiar firewalling problem, traditionally best solved with a mixture of strategies combining packet filters and proxies. We make no claims that TARP Addressing solves any of these complexities, only that there should be cases where firewalls can use the addressing scheme to provide equivalent security without following application level protocols.

12 Limitations

Server facilities using a multi-homed server strategy to serve a variety of domains from the same host [Ste90, p. 93], may not be a good match for TARP's preference to serve from a single fixed address per interface. At the least, they seem incompatible with `inetd`'s wildcard listening sockets and would seem better off if the relevant servers bound only to the the address they are serving. This is more a limitation of BSD, rather than TARP addressing, *per se*.

The security concepts make no contributions to solving problems of inside threats, but this is a recognized limitation of firewalls in general ([Cha92], [CB94]).

Our implementation cannot support more than a single TARP address per interface, and doing so would require extensive kernel modifications. This is for two reasons. First, when faced with an outgoing address decision, the kernel already knows which interface to use, and the implementation determines the Ethernet address of the outgoing interface for use in address computation. Even if we had an extra, valid Ethernet address to use, it would be difficult for the kernel to determine when to assign it. The second set of problems is that all of the problems of multi-homed BSD servers described above would occur.

13 Applicability to other Operating Systems

Although our implementation was built on the UNIX notion of "process groups", it is clearly not necessary to do it that way. Two preconditions are necessary for a substitute mechanism.

First, and most obvious, there has to be some way to generate a 16-bit number not currently in use. Clearly, a counter and an in-use list will suffice. The harder problem is somehow assigning this number to a program or group of related programs. All "related" programs—we define "related" as meaning "they would expect to have the same IP address"—must somehow be linked to this number. The notion of a process group in UNIX captures these semantics quite well; the fact that process groups have the right sort of number is simply a happy accident.

14 Conclusions

We have shown TARP to be a useful new possibility made available by IPv6's Addressing Architecture. Using TARP addresses greatly simplifies firewalling decisions for the machines protecting either clients or servers. Furthermore, by partitioning the address space into client and server addresses and only configuring client network addresses as needed by client activity, it looks to provide particular advantages for hosts acting mostly as network service consumers.

15 Acknowledgments

A number of people involved in the IETF's IPv6 effort (Steve Deering, Thomas Narten, Erik Nordmark, Keith Moore, and others) provided valuable advice. Jon Crowcroft noted that similar facilities could be built on top of Linux's IP Address Masquerading facility. Also, great thanks are due to John Ioannidis for helpful and generous use of the computer laboratory facilties he maintains. We gratefully acknowledge the code, excellent assistance and cheerful bug fixes provided by the *kame* project.

References

[Bel89] Steven M. Bellovin. Security problems in the TCP/IP protocol suite. *Computer Communications Review*, 19(2):32–48, April 1989.

[Bel94] S. Bellovin. On many addresses per host. Request for Comments 1681, Internet Engineering Task Force, August 1994.

[Bel99] Steven M. Bellovin. Distributed firewalls. *;login:*, pages 39–47, November 1999.

[CB94] William R. Cheswick and Steven M. Bellovin. *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison-Wesley, Reading, MA, 1994.

[Cha92] D. Brent Chapman. Network (in)security through IP packet filtering. In *Proceedings of the Third Usenix* UNIX *Security Symposium*, pages 63–76, Baltimore, MD, September 1992.

[Cra99] Matt Crawford. IPv6 node information queries, 1999. Work in progress.

[GS96] Simson Garfinkel and Gene Spafford. *Practical Unix and Internet Security*. O'Reilly, Sebastopol, CA, 1996.

[HC98] D. Harkins and D. Carrel. The internet key exchange (IKE). Request for Comments 2409, Internet Engineering Task Force, November 1998.

[HD98] R. Hinden and S. Deering. IP version 6 addressing architecture. Request for Comments 2373, Internet Engineering Task Force, July 1998.

[Ins97] Institute of Electrical and Electronics Engineers. Guidelines for 64-bit global identifier EUI-64 registration authority, 1997.

[Joh93] M. St. Johns. Identification protocol. Request for Comments 1413, Internet Engineering Task Force, January 1993.

[KA98] S. Kent and R. Atkinson. Security architecture for the internet protocol. Request for Comments 2401, Internet Engineering Task Force, November 1998.

[KAM] http://www.kame.net.

[MDM96] J. McCann, S. Deering, and J. Mogul. Path MTU discovery for IP version 6. Request for Comments 1981, Internet Engineering Task Force, August 1996.

[ND01] T. Narten and R. Draves. Privacy extensions for stateless address autoconfiguration in IPv6. Request for Comments 3041, Internet Engineering Task Force, January 2001.

[NNS98] T. Narten, E. Nordmark, and W. Simpson. Neighbor discovery for IP version 6 (ipv6). Request for Comments 2461, Internet Engineering Task Force, December 1998.

[Pip98] D. Piper. The internet IP security domain of interpretation for ISAKMP. Request for Comments 2407, Internet Engineering Task Force, November 1998.

[Ste90] W. Richard Stevens. UNIX *Network Programming: Networking APIs: Sockets and XTI*, volume 1. Prentice-Hall, Englewood Cliffs, NJ, second edition, 1990.

[TH95] S. Thomson and C. Huitema. DNS extensions to support IP version 6. Request for Comments 1886, Internet Engineering Task Force, December 1995.

[TN98] S. Thomson and T. Narten. IPv6 stateless address autoconfiguration. Request for Comments 2462, Internet Engineering Task Force, December 1998.

[Ven92] Wietse Venema. TCP WRAPPER: Network monitoring, access control and booby traps. In *Proceedings of the Third Usenix* UNIX *Security Symposium*, pages 85–92, Baltimore, MD, September 1992.

[VET$^+$97] P. Vixie, Ed., S. Thomson, Y. Rekhter, and J. Bound. Dynamic updates in the domain name system (DNS UPDATE). Request for Comments 2136, Internet Engineering Task Force, April 1997.

[WS95] Gary R. Wright and W. Richard Stevens. *TCP/IP Illustrated: The Implementation*, volume 2. Addison-Wesley, Reading, MA, 1995.

[Ylo96] Tatu Ylonen. SSH – secure login connections over the internet. In *Proceedings of the Sixth Usenix* UNIX *Security Symposium*, pages 37–42, July 1996.

[ZCC00] Elizabeth D. Zwicky, Simon Cooper, and D. Brent Chapman. *Building Internet Firewalls*. O'Reilly, Sebastopol, CA, second edition, 2000.

Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics

Mark Handley and Vern Paxson
AT&T Center for Internet Research at ICSI (ACIRI)
International Computer Science Institute
Berkeley, CA 94704 USA
{mjh,vern}@aciri.org

Christian Kreibich
Institut für Informatik
Technische Universität München
80290 München, Germany
kreibich@cs.tum.edu

Abstract

A fundamental problem for network intrusion detection systems is the ability of a skilled attacker to *evade* detection by exploiting ambiguities in the traffic stream as seen by the monitor. We discuss the viability of addressing this problem by introducing a new network forwarding element called a traffic *normalizer*. The normalizer sits directly in the path of traffic into a site and patches up the packet stream to eliminate potential ambiguities before the traffic is seen by the monitor, removing evasion opportunities. We examine a number of tradeoffs in designing a normalizer, emphasizing the important question of the degree to which normalizations undermine end-to-end protocol semantics. We discuss the key practical issues of "cold start" and attacks on the normalizer, and develop a methodology for systematically examining the ambiguities present in a protocol based on walking the protocol's header. We then present *norm*, a publicly available user-level implementation of a normalizer that can normalize a TCP traffic stream at 100,000 pkts/sec in memory-to-memory copies, suggesting that a kernel implementation using PC hardware could keep pace with a bidirectional 100 Mbps link with sufficient headroom to weather a high-speed flooding attack of small packets.

1 Introduction

A fundamental problem for network intrusion detection systems (NIDSs) that passively monitor a network link is the ability of a skilled attacker to *evade* detection by exploiting ambiguities in the traffic stream as seen by the NIDS [14]. Exploitable ambiguities can arise in three different ways:

(i) The NIDS may lack complete analysis for the full range of behavior allowed by a particular protocol. For example, an attacker can evade a NIDS that fails to reassemble IP fragments by intentionally transmitting their attack traffic in fragments rather than complete IP datagrams. Since IP end-systems are required to perform fragment reassembly, the attack traffic will still have the intended effect at the victim, but the NIDS will miss the attack because it never reconstructs the complete datagrams.

Of the four commercial systems tested by Ptacek and Newsham in 1998, none correctly reassembled fragments [14].

Also note that an attacker can evade the NIDS even if the NIDS does perform analysis for the protocol (e.g., it does reassemble fragments) if the NIDS's analysis is *incomplete* (e.g., it does not correctly reassemble out-of-order fragments).

(ii) Without detailed knowledge of the victim end-system's protocol implementation, the NIDS may be *unable* to determine how the victim will treat a given sequence of packets if different implementations interpret the same stream of packets in different ways. Unfortunately, Internet protocol specifications do not always accurately specify the complete behavior of protocols, especially for rare or exceptional conditions. In addition, different operating systems and applications implement different subsets of the protocols.

For example, when an end-system receives *overlapping* IP fragments that differ in the purported data for the overlapping region, some end-system's may favor the data first received, others the portion of the overlapping fragment present in the lower fragment, others the portion in the upper fragment.

(iii) Without detailed knowledge of the network topology between the NIDS and the victim end-system, the NIDS may be unable to determine whether a given packet will even be seen by the end-system. For example, a packet seen by the NIDS that has a low Time To Live (TTL) field may or may not have sufficient hop count remaining to make it all the way to the end-system [12]; see below for an example.

If the NIDS believes a packet was received when in fact it did not reach the end-system, then its model of the end-system's protocol state will be incorrect. If the attacker can find ways to systematically ensure that some packets will be received and some not, the attacker may be able to evade the NIDS.

The first of these shortcomings can in principle be addressed by a sufficiently diligent NIDS implementation, making sure that its analysis of each protocol is complete. However, the other two shortcomings are more fundamental: in the absence of external knowledge (end-system implementation details, topology details), *no amount* of analyzer completeness within the NIDS can help it correctly determine the end-system's ultimate processing of the packet stream. On the other hand, the attacker may be able to determine these end-system characteristics for a particular victim by actively probing the victim, perhaps in quite subtle (very hard to detect) ways. Thus, an attacker can craft their traffic so that, whatever algorithms the NIDS analyzer uses, it will err in determining how the end-system behaves.

Figure 1 shows an example of an evasion attack that can exploit either of the last two shortcomings above. The attacker fakes a missing packet, then sends a sequence of TCP packets above the sequence hole that contains the attack, and also sends a sequence of TCP packets containing innocuous data for the same TCP sequence space.

For the moment, ignore the "timed out" packets and assume all of the packets on the left arrive at the victim. Even in this case, the NIDS needs to know precisely how the end-system will interpret the inconsistent "retransmissions"—whether it will use "n" or "r" for

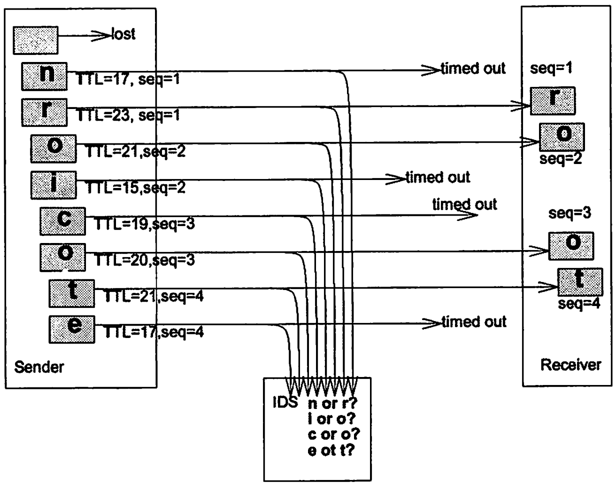


Figure 1: Inconsistent TCP "retransmissions"

sequence #1, "o" or "i" for sequence #2, etc.—when constructing the byte stream presented to the application. Unfortunately, different TCP stacks do different things in this error case; some accept the first packet, and some the second. There is no simple-and-correct rule the NIDS can use for its analysis.

In addition, the attacker may also be able to control which of the packets seen by the NIDS actually arrive at the end-system and which do not. In Figure 1, the attacker does so by manipulating the TTL field so that some of the packets lack sufficient hop count to travel all the way to the victim. In this case, to disambiguate the traffic the NIDS must know exactly how many forwarding hops lie between it and the victim.

One might argue that such evasive traffic or active probing will itself appear anomalous to the NIDS, and therefore the NIDS can detect that an attacker is attempting to evade it. However, doing so is greatly complicated by two factors. First, detection of an attempt at evasion degrades the precision of a NIDS's detection down from identifying the specifics of an attack to only being able to flag that an attack might possibly be in progress. Second, network traffic unfortunately often includes a non-negligible proportion of highly unusual, but benign, traffic, that will often result in *false positives* concerning possible evasion attempts. This is discussed in [12] as the problem of "crud"; examples include inconsistent TCP retransmissions and overlapping inconsistent fragments.

In the above argument we assume the attacker is aware of the existence of the NIDS, has access to its source code (or can deduce the operation of its algorithms) and attack profile database, and that the attacker is actively trying to evade the NIDS. All of these are prudent or plausible assumptions; for example, already the cracker community has discussed the issues [5] and some evasion toolkits (developed by "white hats" to aid in testing and hardening NIDSs) have been developed [2]. Thus, we again emphasize the serious and difficult nature of this problem: unless steps are taken to address all three of the evasion issues discussed above, network intrusion detection based on passive monitoring of traffic will eventually be completely circumventable, and provide no real protection to sites relying on it.

In this paper we consider the viability of addressing the evasion-by-ambiguity problem by introducing a new network forwarding element called a traffic *normalizer*. The normalizer's job is to sit directly in the path of traffic into a site (a "bump in the wire") and patch up or *normalize* the packet stream to remove potential ambiguities. The result is that a NIDS monitoring the normalized traffic stream no longer needs to consider potential ambiguities in interpreting the stream: the traffic as seen by the NIDS is guaranteed unambiguous, thanks to the normalizer. For example, a normalizer processing the traffic shown in Figure 1 might replace the data in any subsequent inconsistent retransmissions with the data from the original version of the same sequence space, so the only text the NIDS (*and the end-system*) would see would be noct.

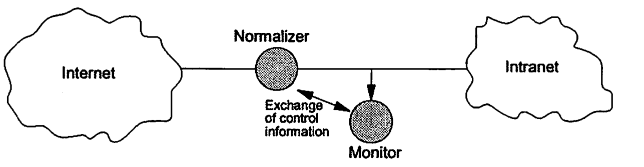


Figure 2: Typical locations of normalizer and NIDS

A normalizer differs from a firewall in that its purpose is not to prevent access to services on internal hosts, but to ensure that access to those hosts takes place in a manner that is unambiguous to the site's NIDS. Figure 2 shows the typical locations of the normalizer relative to the NIDS and the end-systems being monitored. We will refer to traffic traveling from the "Internet" to the "Intranet" as *inbound*, and to traffic in the other direction as *outbound*.

The basic idea of traffic normalization was simultaneously invented in the form of a *protocol scrubber* [8, 13, 17]. The discussion of the TCP/IP scrubber in [8] focuses on ambiguous TCP retransmission attacks like the one described above. The key distinctions between our work and TCP/IP scrubbers is that we attempt to develop a systematic approach to identifying all potential normalizations (we find more than 70, per Appendix A), and we emphasize the implications of various normalizations with regard to maintaining or eroding the end-to-end transport semantics defined by the TCP/IP protocol suite. In addition, we attempt to defend against attacks on the normalizer itself, both through state exhaustion, and through state loss if the attacker can cause the normalizer or NIDS to restart (the "cold start" problem, per § 4.1).

In the next section we discuss other possible approaches for addressing the NIDS ambiguity problem. In § 3 we look at a number of tradeoffs in the design of a normalizer, and in § 4 two important practical considerations. § 5 first presents a systematic approach to discovering possible ambiguities in a protocol as seen by a network analyzer and then applies this approach to analyzing IP version 4. In § 6 we present examples of particularly illuminating normalizations for TCP, including an ambiguity problem that normalization cannot solve. We then discuss in § 7 a user-level normalizer called *norm*, which our performance measurements indicate should be able to process about 100,000 pkts/sec if implemented in the kernel.

2 Other approaches

In this section we briefly review other possible ways of addressing the problem of NIDS evasion, to provide general context for the normalizer approach.

Use a host-based IDS. We can eliminate ambiguities in the traffic stream by running the intrusion detection system (IDS) on all of the end-system hosts rather than by (or in addition to) passively monitoring network links. As the host IDS has access to the protocol state *above* the IP and transport stacks, it has unambiguous information as to how the host processes the packet stream. However, this approach is tantamount to giving up on network intrusion detection, as it loses the great advantage of being able to provide monitoring for an entire site cheaply, by deploying only a few monitors to watch key network links. Host-based systems also potentially face major deployment and management difficulties. In this work, we are concerned with the question of whether purely network-based IDS's can remain viable, so we do not consider this solution further.

Understand the details of the intranet. In principle, a NIDS can eliminate much of the ambiguity if it has access to a sufficiently rich database cataloging the particulars of all of the end-system protocol implementations and the network topology. A major challenge with this approach is whether we can indeed construct such a database, particularly for a large site. Perhaps adding an active probing element to a NIDS can do so, and there has been some initial work in this regard [9]. However, another difficulty is that the NIDS would need to know how to make use of the database—it would require a model of every variant of every OS and application running within the site, potentially an immense task.

Bifurcating analysis. Finally, in some cases the NIDS can employ *bifurcating analysis* [12]: if the NIDS does not know which of two possible interpretations the end-system may apply to incoming packets, then it splits its analysis context for that connection into multiple threads, one for each possible interpretation, and analyzes each context separately from then onwards.

Bifurcating analysis works well when there are only a small number of possible interpretations no matter how many packets are sent. An example would be in the interpretation of the BACKSPACE vs. DELETE character during the authentication dialog at the beginning of a Telnet connection (before the user has an opportunity to remap the meaning of the characters): generally, either one or the other will delete the character of text most recently typed by the user. The NIDS can form two contexts, one interpreting DELETE as the deletion character, and the other interpreting BACKSPACE as the deletion character. Since the end-system will be in one state or the other, one of the analysis contexts will be correct at the NIDS no matter how many packets are sent.

However, bifurcating analysis will not be suitable if each arriving ambiguous packet requires an additional bifurcation, as in this case an attacker (or an inadvertent spate of "crud") can send a stream of packets such that the number of analysis contexts explodes exponentially, rapidly overwhelming the resources of the NIDS. Consider, for example, the attack shown in Figure 1. If the NIDS bifurcates its analysis on receipt of each potentially ambiguous packet, it will rapidly require a great deal of state and many analysis threads. Once it has seen the eight packets shown, it will need threads for the possible text root, nice, rice, noot, niot, roce, roct, *etc.* ...

3 Normalization Tradeoffs

When designing a traffic normalizer, we are faced with a set of tradeoffs, which can be arranged along several axes:

- extent of normalization vs. protection
- impact on end-to-end semantics (service models)
- impact on end-to-end performance
- amount of state held
- work offloaded from the NIDS

Generally speaking, as we increase the degree of normalization and protection, we need to hold more state; performance decreases both for the normalizer and for end-to-end flows; and we impact end-to-end semantics more. Our goal is not to determine a single "sweet spot," but to understand the character of the tradeoffs, and, ideally, design a system that a site can tune to match their local requirements.

Normalization vs. protection. As a normalizer is a "bump in the wire," the same box performing normalization can also perform firewall functionality. For example, a normalizer can prevent known attacks, or shut down access to internal machines from an external host when the NIDS detects a probe or an attack. In this paper we concentrate mainly on normalization functionality, but will occasionally discuss protective functionality for which a normalizer is well suited.

End-to-end semantics. As much as possible, we would like a normalizer to preserve the end-to-end semantics of well-behaved network protocols, whilst cleaning up misbehaving traffic. Some packets arriving at the normalizer simply cannot be correct according to the protocol specification, and for these there often is a clear normalization to apply. For example, if two copies of an IP fragment arrive with the same fragment offset, but containing different data, then dropping either of the fragments or dropping the whole packet won't undermine the correct operation of the particular connection. Clearly the operation was already incorrect.

However, there are other packets that, while perfectly legal according to the protocol specifications, may still cause ambiguities for the NIDS. For example, it is perfectly legitimate for a packet to arrive at the normalizer with a low TTL. However, per the discussion in the Introduction, the NIDS cannot be sure whether the packet will reach the destination. A possible normalization for such packets is to increase its TTL to a large value.[1] For most traffic, this will have no adverse effect, but it will break diagnostics such as `traceroute`, which rely on the semantics of the TTL field for their correct operation.

Normalizations like these, which erode but do not brutally violate the end-to-end protocol semantics, present a basic tradeoff that each site must weigh as an individual policy decision, depending on its user community, performance needs, and threat model. In our analysis of different normalizations, we place particular emphasis on this tradeoff, because we believe the long-term utility of preserving end-to-end semantics is often underappreciated and at risk of being sacrificed for short-term

[1] Clearly, this is dangerous unless there is no possibility of the packet looping around to the normalizer again.

expediency.

Impact on end-to-end performance. Some normalizations are performed by modifying packets in a way that removes ambiguities, but also adversely affects the performance of the protocol being normalized. There is no clear answer as to how much impact on performance might be acceptable, as this clearly depends on the protocol, local network environment, and threat model.

Stateholding. A NIDS system must hold state in order to understand the context of incoming information. One form of attack on a NIDS is a *stateholding attack*, whereby the attacker creates traffic that will cause the NIDS to instantiate state (see § 4.2 below). If this state exceeds the NIDS's ability to cope, the attacker may well be able to launch an attack that passes undetected. This is possible in part because a NIDS generally operates passively, and so "fails open."

A normalizer also needs to hold state to correct ambiguities in the data flows. Such state might involve keeping track of unacknowledged TCP segments, or holding IP fragments for reassembly in the normalizer. However, unlike the NIDS, the normalizer is in the forwarding path, and so has the capability to "fail closed" in the presence of stateholding attacks. Similarly, the normalizer can perform "triage" amongst incoming flows: if the normalizer is near state exhaustion, it can shut down and discard state for flows that do not appear to be making progress, whilst passing and normalizing those that do make progress. The assumption here is that without complicity from internal hosts (see below), it is difficult for an attacker to fake a large number of *active* connections and stress the normalizer's stateholding.

But even given the ability to perform triage, if a normalizer operates fail-closed then we must take care to assess the degree to which an attacker can exploit stateholding to launch a denial-of-service attack against a site, by forcing the normalizer to terminate some of the site's legitimate connections.

Inbound vs. outbound traffic. The design of the Bro network intrusion detection system assumes that it is monitoring a bi-directional stream of traffic, and that either the source or the destination of the traffic can be trusted [12]. However it is equally effective at detecting inbound or outbound attacks.

The addition of a normalizer to the scenario potentially introduces an asymmetry due to its location—the normalizer protects the NIDS against ambiguities by processing the traffic before it reaches the NIDS (Figure 2). Thus, an internal host attempting to attack an external host might be able to exploit such ambiguities to evade the local NIDS. If the site's threat model includes such attacks, either two normalizers may be used, one on either side of the NIDS, or a NIDS integrated into a single normalizer. Finally, we note that if both internal and external hosts in a connection are compromised, there is little any NIDS or normalizer can do to prevent the use of encrypted or otherwise covert channels between the two hosts.

Whilst a normalizer will typically make most of its modifications to incoming packets, there may also be a number of normalizations it applies to outgoing packets. These normalizations are to ensure that the internal and external hosts' protocol state machines stay in step despite the normalization of the incoming traffic. It is also possible to normalize outgoing traffic to prevent unintended information about the internal hosts from escaping ([17], and see § 5.1 below).

Protection vs. offloading work. Although the primary purpose of a normalizer is to prevent ambiguous traffic from reaching the NIDS where it would either contribute to a state explosion or allow evasion, a normalizer can also serve to offload work from the NIDS. For example, if the normalizer discards packets with bad checksums, then the NIDS needn't spend cycles verifying checksums.

4 Real-world Considerations

Due to the adversarial nature of attacks, for security systems it is particularly important to consider not only the principles by which the system operates, but as much as possible also the "real world" details of operating the system. In this section, we discuss two such issues, the "cold start" problem, and attackers targeting the normalizer's operation.

4.1 Cold start

It is natural when designing a network traffic analyzer to structure its analysis in terms of tracking the progression of each connection from the negotiation to begin it, through the connection's establishment and data transfer operations, to its termination. Unless carefully done, however, such a design can prove vulnerable to incorrect analysis during a *cold start*. That is, when the analyzer first begins to run, it is confronted with traffic from already-established connections for which the analyzer lacks knowledge of the connection characteristics negotiated when the connections were established.

For example, the TCP scrubber [8] requires a connection to go through the normal start-up handshake. However, if a valid connection is in progress, and the scrubber restarts or otherwise loses state, then it will terminate any connections in progress at the time of the cold start, since to its analysis their traffic exchanges appear to violate the protocol semantics that require each newly seen connection to begin with a start-up handshake.

The cold-start problem also affects the NIDS itself. If the NIDS restarts, the loss of state can mean that previously monitored connections are no longer monitorable because the state negotiated at connection setup time is no longer available. As we will show, techniques required to allow clean normalizer restarts can also help a NIDS with cold start (§ 6.2).

Finally, we note that cold start is not an unlikely "corner case" to deal with, but instead an on-going issue for normalizers and NIDS alike. First, an attacker might be able to force a cold start by exploiting bugs in either system. Second, from operational experience we know that one cannot avoid occasionally restarting a monitor system, for example to reclaim leaked memory or update configuration files. Accordingly, a patient attacker who keeps a connection open for a long period of time can ensure a high probability that it will span a cold start.

4.2 Attacking the Normalizer

Inevitably we must expect the normalizer itself to be the target of attacks. Besides complete subversion, which can be prevented only though good design and coding practice, two other ways a normalizer can be attacked are stateholding attacks and CPU overload attacks.

Stateholding attacks. Some normalizations are stateless. For example, the TCP MSS option (Maximum Segment Size) is only allowed in TCP SYN packets. If a normalizer sees a TCP packet with an MSS Option but no SYN flag, then this is illegal; but even so, it may be unclear to the NIDS what the receiving host will do with the option, since its TCP implementation might incorrectly still honor it. Because the use of the option is illegal, the normalizer can safely remove it (and adjust the TCP checksum) without needing to instantiate any state for this purpose.

Other normalizations require the normalizer to hold state. For example, an attacker can create ambiguity by sending multiple copies of an IP fragment with different payloads. While a normalizer can remove fragment-based ambiguities by reassembling all fragmented IP packets itself before forwarding them (and if necessary re-fragmenting correctly), to do this, the normalizer must hold fragments until they can be reassembled into a complete packet. An attacker can thus cause the normalizer to use up memory by sending many fragments of packets without ever sending enough to complete a packet.

This particular attack is easily defended against by simply bounding the amount of memory that can be used for fragments, and culling the oldest fragments from the cache if the fragment cache fills up. Because fragments tend to arrive together, this simple strategy means an attacker has to flood with a very high rate of fragments to cause a problem. Also, as IP packets are unreliable, there's no guarantee they arrive anyway, so dropping the occasional packet doesn't break any end-to-end semantics.

More difficult to defend against is an attacker causing the normalizer to hold TCP state by flooding in, for example, the following ways:

1. Simple SYN flooding with SYNs for multiple connections to the same or to many hosts.
2. ACK flooding. A normalizer receiving a packet for which it has no state might be designed to then instantiate state (in order to address the "cold start" problem).
3. Initial window flooding. The attacker sends a SYN to a server that exists, receives a SYN-ACK, and then floods data without waiting for a response. A normalizer would normally temporarily store unacknowledged text to prevent inconsistent retransmissions.

Our strategy for defending against these is twofold. First, the normalizer knows whether or not it's under attack by monitoring the amount of memory it is consuming. If it's not under attack, it can instantiate whatever state it needs to normalize correctly. If it believes it's under attack, it takes a more conservative strategy that is designed to allow it to survive, although some legitimate traffic will see degraded performance.

In general our aim when under attack is to only instantiate TCP connection state when we see traffic from an internal (and hence trusted) host, as this restricts stateholding attacks on the normalizer to those actually involving real connections to internal hosts. Note here that the normalizer is explicitly *not* attempting to protect the internal hosts from denial-of-service attacks; only to protect itself and the NIDS.

CPU overload attacks. An attacker may also attempt to overload the CPU on the normalizer. However, unlike stateholding attacks, such an attack cannot cause the normalizer to allow an ambiguity to pass. Instead, CPU overload attacks can merely cause the normalizer to forward packets at a slower rate than it otherwise would.

In practice, we find that most normalizations are rather cheap to perform (§ 7.2), so such attacks need to concentrate on the normalizations where the attacker can utilize computational complexity to their advantage. Thus, CPU utilization attacks will normally need to be combined with stateholding attacks so that the normalizer performs an expensive search in a large state-space. Accordingly, we need to pay great attention to the implementation of such search algorithms, with extensive use of constant-complexity hash algorithms to locate matching state. An additional difficulty that arises is the need to be opportunistic about garbage collection, and to apply algorithms that are low cost at the possible expense of not being completely optimal in the choice of state that is reclaimed.

5 A Systematic Approach

For a normalizer to completely protect the NIDS, in principle we must be able to normalize every possible sequence of packets that the NIDS might treat differently from the end-system. Given that the NIDS cannot possibly know all the application state at the end-system for all applications, we focus in this work on the more tractable problem of normalizing the internetwork (IP, ICMP) and transport (TCP, UDP) layers.

Even with this somewhat more restricted scope, we find there are still a very large number of possible protocol ambiguities to address. Consequently, it behooves us to develop a systematic methodology for attempting to identify and analyze all of the possible normalizations. The methodology we adopt is to walk through the packet headers of each protocol we consider. This ensures that we have an opportunity to consider each facet of the semantics associated with the protocol.

For each header element, we consider its possible range of values, their semantics, and ways an attacker could exploit the different values; possible actions a normalizer might take to thwart the attacks; and the effects these actions might have on the protocol's end-to-end semantics. Whilst our primary intention is to explore the possible actions a normalizer can take, the exercise also raises interesting questions about the incompleteness of the specifications of error handling behavior in Internet protocols, and about the nature of the intentional and unintentional end-to-end semantics of Internet protocols.

For reasons of space, we confine our analysis here to a single protocol; we pick IP (version 4) because it is simple enough to cover fairly thoroughly in this paper, yet has rich enough semantics (especially fragmentation) to convey the flavor of more complicated normalizations. In § 6 we then present some particularly illuminating examples of TCP normalizations. We defer our methodical analysis of TCP (and UDP and ICMP) to [4].

Note that many of the normalizations we discuss below appear to address very unlikely evasion scenarios. However, we believe the right design approach is to normalize *everything* that we can see how to correctly normalize, because packet manipulation and semantic ambiguity is sufficiently subtle that we may miss an attack, but still thwart it because we normalized away the degrees of freedom to express the attack.

Figure 3 shows the fields of the IP packet header. For each field we identify possible issues that need normalization and discuss the effects of our solutions on end-to-end semantics. The reader preferring to delve into only more interesting normalizations may choose to jump ahead to § 5.1.

Version. A normalizer should only pass packets with IP version fields which the NIDS understands.

Header length. It may be possible to send a packet with an incorrect header length field that arrives at an end-systems and is accepted. However, other operating systems or internal routers may discard the packet. Thus the NIDS does not know if the packet will be processed or not.
Solution: If the header length field is less than 20 bytes, the header is incomplete, and the packet should be discarded. If the header length field exceeds the packet length, the packet should be discarded. (See *Total length* below for a discussion of exactly what constitutes the packet length.)
Effect on semantics: Packet is ill-formed—no adverse effect.
Note: If the header length is greater than 20 bytes, this indicates options are present. See IP option processing below.

Type Of Service/Diffserv/ECN. These bits have recently been reassigned to differentiated services [11] and explicit congestion notification [15].

Issue: The Diffserv bits might potentially be used to deterministically drop a subset of packets at an internal Diffserv-enabled router, for example by sending bursts of packets that violate the conditioning required by their Diffserv class.
Solution: If the site does not actually use Diffserv mechanisms for incoming traffic, clear the bits.
Effect on semantics: If Diffserv is not being used internally, the bits should be zero anyway, so zeroing them is safe. Otherwise, clearing them breaks use of Diffserv.

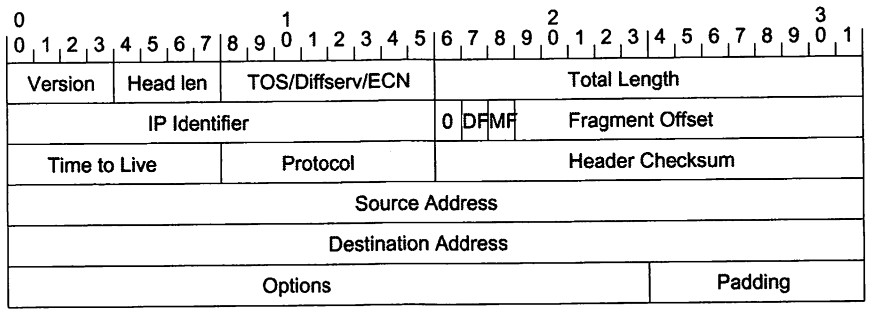


Figure 3: IP v4 Header

Issue: Some network elements (such as firewalls) may drop packets with the ECN bits set, because they are not yet cognizant of the new interpretation of these header bits.
Solution: Clear the bits for all connections unless the connection previously negotiated use of ECN. Optionally, remove attempts to negotiate use of ECN.
Effect on semantics: For connections that have not negotiated use of ECN, no end-to-end effect. Removing attempted negotiation of ECN will prevent connections from benefiting from avoiding packet drops in some circumstances.

Total length. If the total length field does not match the actual total length of the packet as indicated by the link layer, then some end-systems may treat the packet as being one length, some may treat it as being the other, and some may discard the packet.
Solution: Discard packets whose length field exceeds their link-layer length. Trim packets with longer link-layer lengths down to just those bytes indicated by the length field.
Effect on semantics: None, only ill-formed packets are dropped.

IP Identifier. See § 5.1.

Must Be Zero. The IP specification requires that the bit between "IP Identifier" and "DF" must be zero.
Issue: If the bit is set, then intermediary or end-systems processing the packet may or may not discard it.
Solution: Clear the bit to zero.
Effect on semantics: None, since the bit is already required to be zero.
Note: We might think that we could just as well discard the packet, since it violates the IP specification. The benefit of merely clearing the bit is that if in the future a new use for the bit is deployed, then clearing the bit will permit connections to continue, rather than abruptly terminating them, in keeping with the philosophy that Internet protocols should degrade gradually in the presence of difficulties.

Don't Fragment (DF) flag. If DF is set, and the Maximum Transmission Unit (MTU) anywhere in the internal network is smaller than the MTU on the access link to the site, then an attacker can deterministically cause some packets to fail to reach end-systems behind the smaller MTU link. This is done by setting DF on packets with a larger MTU than the link.
Note: The NIDS might be able to infer this attack from ICMP responses sent by the router that drops the packets, but the NIDS needs to hold state to do so, leading to state-holding attacks on the NIDS. Also, it is not certain that the NIDS will always see the ICMP responses, due to rate-limiting and multi-pathing.
Solution: Clear DF on incoming packets.
Effect on semantics: Breaks "Path MTU Discovery." If an incoming packet is too large for an internal link, it will now be fragmented, which could have an adverse effect on performance—in the router performing the fragmentation, in the end host performing reassembly, or due to increased effective packet loss rates on the network links after fragmentation occurs [6]. That said, for many network environments, these are unlikely to be serious problems.

Issue: Packets arriving with DF set and a non-zero fragmentation offset are illegal. However, it is not clear whether the end-system will discard these packets.
Solution: Discard such packets.
Effect on semantics: None, ill-formed packet.

More Fragments (MF) flag, **Fragment Offset.** We treat these two fields together because they are interpreted together. An ambiguity arises if the NIDS sees two fragments that overlap each other and differ in their contents. As noted in [14], different operating systems resolve the ambiguity differently.
Solution: Reassemble incoming fragments in the normalizer rather than forwarding them. If required, re-fragment the packet for transmission to the internal network if it is larger than the MTU.
Effect on semantics: Reassembly is a valid operation for a router to perform, although it is not normally done. Thus this does not affect end-to-end semantics.
Note: A normalizer that reassembles fragments is vulnerable to stateholding attacks, and requires an appropriate triage strategy to discard partially reassembled packets if the normalizer starts to run out of memory.

Issue: Packets where the length plus the fragmentation offset exceeds 65535 are illegal. They may or may not be accepted by the end host. They may also cause some hosts to crash.
Solution: Drop the packets.
Effect on semantics: Packet is ill-formed, so no effect.

TTL (Time-to-live). As with DF, an attacker can use TTL to manipulate which of the packets seen by the NIDS reaches the end-system, per the discussion for Figure 1.
Solution #1: In principle, a NIDS could measure the number of hops to every end host, and ignore packets that lack sufficient TTL. In practice, though, at many sites this requires holding a large amount of state, and it is possible that the internal routing may change (possibly triggered by the attacker in some way) leaving a window of time where the NIDS's measurement is incorrect.
Solution #2: The NIDS may also be able to see ICMP time-exceeded-in-transit packets elicited by the attack. However, ICMP responses are usually rate limited, so the NIDS may still not be able to tell exactly which packets were discarded.
Solution #3: Configure the normalizer with a TTL that is larger than the longest path across the internal site. If packets arrive that have a TTL lower than the configured minimum, then the normalizer restores the TTL to the minimum.
Effect on semantics: First, if a routing loop passes through the normalizer, then it may be possible for packets to loop forever, rapidly consuming the available bandwidth. Second, restoring TTL will break `traceroute` due to its use of limited-TTL packets to discover forwarding hops. Third, restoring TTL on multicast packets may impair the performance of applications that use expanding ring searches. The effect will be that all internal hosts in the group appear to be immediately inside the normalizer from the point of view of the search algorithm.

Protocol. The protocol field indicates the next-layer protocol, such as TCP or UDP. Blocking traffic based on it is a firewall function and not a normalizer function. However, an administrator may still configure a normalizer to discard packets that do not contain well-known protocols, such as those the NIDS understands.

IP header checksum. Packets with incorrect IP header checksums might possibly be accepted by end-hosts with dodgy IP implementations.
Solution: In practice this is not a likely scenario, but the normalizer can discard these packets anyway, which avoids the NIDS needing to verify checksums itself.
Effect on semantics: Normally, no effect. However, it might be possible to use corrupted packets to gather information on link errors or to signal to TCP not to back off because the loss is due to corruption and not congestion. But since routers will normally discard packets with incorrect IP checksums anyway, the issue is likely moot.

Source address. If the source address of an IP packet is invalid in some way, then the end-host may or may not discard the packet. Examples are `127.0.0.1` (localhost), `0.0.0.0` and `255.255.255.255` (broadcast), multicast (class D) and class E addresses.
Solution: Drop the packet.
Effect on semantics: None, packet is ill-formed.
Note: If the incoming packet has a source address belonging to a known internal network, the normalizer might be configured to drop the packet. This is more firewall-type functionality than normalization, but will generally be desirable. However it would break applications that rely on "source routing" packets via an external host and back into the site, such as using `traceroute` to trace a route from an external site back *into* the tracing site. Also, if an outgoing packet has a source address that does not belong to a known internal network, the normalizer might be configured to drop the packet.

Destination address. Like source addresses, invalid destination addresses might cause unexpected behavior at internal hosts. Examples are local broadcast addresses ("smurf" attacks), the localhost and broadcast addresses mentioned above, and class E addresses (which are currently unused).
Solution: Drop the packet. In addition, the normalizer should be capable of dropping incoming packets with destination addresses that would not normally be routed to the site; these might appear as a result of source-routing, and it is unclear what effect they might have on internal hosts or routers.
Effect on semantics: None, destination is illegal.

IP options. IP packets may contain IP options that modify the behavior of internal hosts, or cause packets to be interpreted differently.
Solution: Remove IP options from incoming packets.
Effect on semantics: For end-to-end connections, presumably none, as IP options should not have effects visible at higher layers; *except* the absence of an option may impair end-to-end connectivity, for example because the connectivity requires source routing. For diagnostics tools, potentially serious.
That said, the reality today is that options generally suffer from poor performance because routers defer their processing to the "slow path," and many sites disable their use to counter certain security risks. So in practice, removing IP options should have little ill effect, other than the loss of source routing for diagnosing connectivity problems. This last can be addressed (as can all semantic tradeoffs associated with normalization) through site-specific policies controlling the normalizer's operation.

Padding. The padding field at the end of a list of IP options is explicitly ignored by the receiver, so it is difficult to see that it can be manipulated in any useful way. While it does provide a possible covert channel, so do many other header fields, and thwarting these is not a normalizer task.
Solution: Zero the padding bytes, on the principle that we perform normalizations even when we do not know of a corresponding attack.
Effect on semantics: None, field is explicitly ignored.

5.1 The IP Identifier and Stealth Port Scans

The IP identifier (ID) of outgoing packets may give away information about services running on internal hosts. This issue is not strictly a normalizer problem, but the normalizer is in a location well suited to deal with the issue.

One particular problem is the exceedingly devious stealth port-scanning technique described in [16, 18], which enables an attacker to probe the services running on a remote host without giving away the IP address of the host being used to conduct the scan. Figure 4 illustrates the technique, which we review here to develop how a normalizer can thwart it. Host A is the attacker, V is the victim, and P is the *patsy*. The patsy must run an operating system that increments the IP ID by one[2] for every packet sent, no matter to what destination—many common operating systems use such a "global" IP ID.

Host A continually exchanges packets with host P, either through a TCP transfer or simply by pinging it. While doing this, the IP IDs of the responses from P to A normally increment by one from one packet to the next. Now A fakes a TCP SYN to the port on V they wish to probe, and they fake the source address of the packet as being from P.

If there is no service listening on the port, V sends a RST to P. As P has no associated connection state, P ignores the RST, and there is no effect on the IP IDs of the stream of packets from P to A.

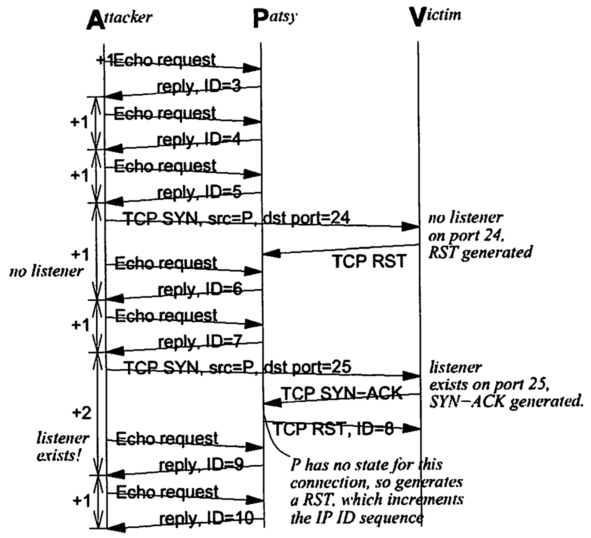


Figure 4: Stealth Port Scan

However, if there is a service listening on the port, V sends a SYN-ACK to P to complete the connection, rather than a RST. P has no state for this connection, and promptly sends a RST back to V. In so doing, the global ID sequence on P increases by one; consequently, in the stream of packets from P to A, the attacker observes a step of two (rather than one) in the ID sequence, since it missed one of the packets sent by P, namely the RST from P to V.

Thus P and not A appears to be the host conducting the port-scan, whereas in fact it is completely innocent. V never sees a packet with a source of A. If A chooses a different patsy for every port it wishes to check, then this port scan is very hard to detect.

The solution for patsies is for the normalizer to scramble (in a cryptographically secure, but reversible fashion) the IP IDs of incoming and outgoing packets. This prevents internal hosts from being used as patsies for such scans. The effect on semantics is that any diagnostic protocol that reports the IP IDs of incoming packets back to the sender may break. ICMP messages can still function if the normalizer applies the unscrambling to the embedded ID fields they carry.

[2]More generally, advances the ID field in a predictable fashion.

The solution for victims is to use the "reliable RST" technique (see § 6.1 below). The normalizer transmits a "keep-alive" acknowledgment (ACK) packet behind every RST packet it forwards out of the site. When the ACK arrives at the patsy, the patsy will reply with a RST, just as it does in the SYN-ACK case. Consequently, the IP ID sequence as seen by the attacker will jump by two in both cases, whether the victim is running the given service or not.

Sending keep-alives for reliable RSTs generates extra traffic, but has no effect on end-to-end semantics, since the keep-alive ACK following the RST is guaranteed to be either rejected by the victim (if it first received the RST) or ignored (if the RST was lost and the connection remains open).

6 Examples of TCP Normalizations

We applied the same "walk the header" methodology as in the previous section to TCP, UDP, and ICMP. However, due to space limitations we defer the detailed analysis to [4], and in this section focus on three examples for TCP that illuminate different normalization issues: reliable RSTs, cold start for TCP, and an example of a TCP ambiguity that a normalizer cannot remove.

6.1 Reliable RSTs

With TCP, the control signals for connection establishment and completion (SYN and FIN, respectively) are delivered reliably, but the "abrupt termination" (RST) signal is not. This leads to a significant problem: in general, both a normalizer and a NIDS needs to tear down state for an existing connection once that connection completes, in order to recover the associated memory. But it is not safe to do so upon seeing a RST, because the RST packet might be lost prior to arriving at the receiver, or might be rejected by the receiver.

Thus, a monitor cannot tell whether a given RST does in fact terminate its corresponding connection. If the monitor errs and assumes it does when in fact it did not, then an attacker can later continue sending traffic on the connection, and the monitor will lack the necessary state (namely, that the connection is still established, and with what sequence numbers, windows, etc.) to correctly interpret that traffic. On the other hand, if the monitor assumes the RST does *not* terminate the connection, then it is left holding the corresponding state potentially indefinitely. (Unfortunately, RST-termination is not uncommon in practice, so even for benign traffic, this state will grow significantly over time.)

The RST might fail to arrive at the receiver because of normal loss processes such as buffer overflows at congested routers, or because of manipulation by an attacker, such as the TTL games discussed in the context of Figure 1. In addition, the rules applied by receivers to determine whether a particular RST is valid vary across different operating systems, which the NIDS likely cannot track.

A general solution to this problem would be to ensure that RSTs are indeed delivered and accepted, i.e., we want "reliable RSTs." We can do so, as follows. Whenever the normalizer sees a RST packet sent from A to B, after normalizing it and sending it on, it synthesizes a second packet and sends that to B, too. This additional packet takes the form of a TCP "keep-alive," which is a dataless[3] ACK packet with a sequence number just below the point cumulatively acknowledged by B. The TCP specification requires that B must in turn reply to the keep-alive with an ACK packet of its own, one with the correct sequence number to be accepted by A, to ensure that the two TCP peers are synchronized. However, B only does this if the connection is still open; if it is closed, it sends a RST in response to the keep-alive.

Thus, using this approach, there are four possible outcomes whenever the normalizer forwards a RST packet (and the accompanying keep-alive):

(i) The RST was accepted by B, and so B will generate another RST back to A upon receipt of the keep-alive;

(ii) the RST either did not make it to B, or B ignored it, in which case B will generate an ACK in response to the keep-alive;

(iii) the keep-alive did not make it to B, or B ignored it (though this latter shouldn't happen);

(iv) or, the response B sent in reply to the keep-alive was lost before the normalizer could see it.

The normalizer then uses the following rule for managing its state upon seeing a RST: *upon seeing a RST from A to B, retain the connection state; but subsequently, upon seeing a RST from B to A, tear down the state.*[4] Thus, the normalizer only discards the connection state upon seeing proof that B has indeed terminated the connection. Note that if either A or B misbehaves, the scheme still works, because one of the RSTs will still

[3] In practice, one sends the last acknowledged byte if possible, for interoperability with older TCP implementations.

[4] Of course we do not send a keep-alive to make the second RST reliable or we'd initiate a RST war.

have been legitimate; only if A and B collude will the scheme fail, and, as noted earlier, in that case there is little a normalizer or a NIDS can do to thwart evasion.

The rule above addresses case *(i)*. For case *(ii)*, the normalizer needn't do anything special (it still retains the connection state, in accordance with the rule). For cases *(iii)* and *(iv)*, it will likewise retain the state, perhaps needlessly; but these cases should be rare, and are not subject to manipulation by A. They could be created by B if B is malicious; but not to much effect, as in that case the connection is already terminated as far as A is concerned.

6.2 Cold start for TCP

Recall that the "cold start" problem concerns how a normalizer should behave when it sees traffic for a connection that apparently existed before the normalizer began its current execution (§ 4.1). One particular goal is that the normalizer (and NIDS) *refrain from instantiating state* for apparently-active connections unless they can determine that the connection is indeed active; otherwise, a flood of bogus traffic for a variety of non-existent connections would result in the normalizer creating a great deal of state, resulting in a state-holding attack.

Accordingly, we need some way for the normalizer to distinguish between genuine, pre-existing connections, and bogus, non-existent connections, and to do so in a *stateless* fashion!

As with the need in the previous section to make RSTs trustworthy, we can again use the trick of encapsulating the uncertainty in a probe packet and using the state held (or not held) at the receiver to inform the normalizer's decision process.

Our approach is based on the assumption that the normalizer lies between a trusted network and an untrusted network, and works as follows. Upon seeing a packet from A to B for which the normalizer does not have knowledge of an associated connection, if A is from the trusted network, then the normalizer instantiates state for a corresponding connection and continues as usual. However, if A is from the untrusted network, the normalizer transforms the packet into a "keep-alive" by stripping off the payload and decrementing the sequence number in the header. It then forwards the modified packet to B and forgets about it. If there is indeed a connection between A and B, then B will respond to the keep-alive with an ACK, which will suffice to then instantiate state for the connection, since B is from the trusted network. If no connection does in fact exist, then B will either respond with a RST, or not at all (if B itself does not exist, for example). In both of these cases, the normalizer does not wind up instantiating any state.

The scheme works in part because TCP is reliable: removing the data from a packet does not break the connection, because A will work diligently to eventually deliver the data and continue the connection.

(Note that a similar scheme can also be applied when the normalizer sees an initial SYN for a new connection: by only instantiating state for the connection upon seeing a SYN-ACK from the trusted network, the load on a normalizer in the face of a SYN flooding attack is diminished to reflect the rate at which the target can absorb the flood, rather than the full incoming flooding rate.)

Even with this approach, though, cold start for TCP still includes some subtle, thorny issues. One in particular concerns the *window scaling* option that can be negotiated in TCP SYN packets when establishing a connection. It specifies a left-shift to be applied to the 16 bit window field in the TCP header, in order to permit receiver windows of greater than 64 KB. In general, a normalizer must be able to tell whether a packet will be accepted at the receiver. Because receivers can discard packets with data that lies beyond the bounds of the receiver window, the normalizer needs to know the window scaling factor in order to mirror this determination. However, upon cold start, the normalizer cannot determine the window scaling value, because the TCP endpoints no longer exchange it, they just use the value they agreed upon at connection establishment.

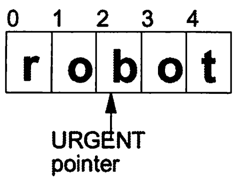
We know of no fully reliable way by which the normalizer might infer the window scaling factor in this case. Consequently, if the normalizer wishes to avoid this ambiguity, it must either ensure that window scaling is simply not used, i.e., *it must remove the window scale option from all TCP SYN packets* to prevent its negotiation (or it must have access to persistent state so it can recover the context for each active connection unambiguously).

Doing so is not without a potentially significant cost: window scaling is required for good performance for connections operating over long-haul, high-speed paths [1], and sites with such traffic might in particular want to disable this normalization.

More generally, this aspect of the cold start problem illustrates how normalizations can sometimes come quite expensively. The next section illustrates how they are sometimes simply not possible.

6.3 Incompleteness of Normalization

In the absence of detailed knowledge about the various applications, normalizations will tend to be restricted to the internetwork and transport layers. However, even at the transport level a normalizer cannot remove all possible ambiguities. For example, the semantics of the TCP urgent pointer cannot be understood without knowing the semantics of the application using TCP:



If the sender sends the text "`robot`" with the TCP urgent pointer set to point to the letter "`b`", then the application may receive *either* "`robot`" or "`root`," depending on the socket options enabled by the receiving application. Without knowledge of the socket options enabled, the normalizer cannot correctly normalize such a packet because either interpretation of it could be valid.

In this case, the problem is likely not significant in practice, because all protocols of which we are aware either enable or disable the relevant option for the entire connection—so the NIDS can use a bifurcating analysis without the attacker being able to create an exponential increase in analysis state. However, the example highlights that normalizers, while arguably very useful for reducing the evasion opportunities provided by ambiguities, are not an all-encompassing solution.

7 Implementation

We have implemented *norm*, a fairly complete, user-level normalizer for IP, TCP, UDP and ICMP. The code comprises about 4,800 lines of C and uses `libpcap` [10] to capture packets and a raw socket to forward them. We have currently tested *norm* under FreeBSD and Linux, and will release it (and NetDuDE, see below) publicly in Summer 2001 via *www.sourceforge.net.*

Naturally, for high performance a production normalizer would need to run in the kernel rather than at user level, but our current implementation makes testing, debugging and evaluation much simpler.

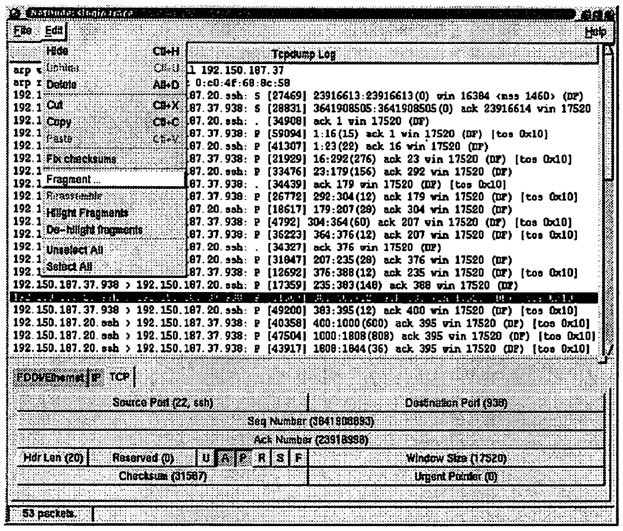
Appendix A summarizes the complete list of normalizations *norm* performs, and these are discussed in detail in [4]. Here we describe our process for testing and evaluating *norm*, and find that the performance on commodity PC hardware is adequate for deployment at a site like ours with a bidirectional 100Mb/s access link to the Internet.

7.1 Evaluation methodology

In evaluating a normalizer, we care about completeness, correctness, and performance. The evaluation presents a challenging problem because by definition most of the functionality of a normalizer applies only to unusual or "impossible" traffic, and the results of a normalizer in general are invisible to connection endpoints (depending on the degree to which the normalizations preserve end-to-end semantics). We primarily use a trace-driven approach, in which we present the normalizer with an input trace of packets to process as though it had received them from a network interface, and inspect an output trace of the transformed packets it in turn would have forwarded to the other interface.

Each individual normalization needs to be tested in isolation to ensure that it behaves as we intend. The first problem here is to obtain test traffic that exhibits the behavior we wish to normalize; once this is done, we need to ensure that *norm* correctly normalizes it.

With some anomalous behavior, we can capture packet traces of traffic that our NIDS identifies as being ambiguous. Primarily this is "crud" and not real attack traffic [12]. We can also use tools such as *nmap* [3] and *fragrouter* [2] to generate traffic similar to that an attacker might generate. However, for most of the normalizations we identified, no real trace traffic is available, and so we must generate our own.



```
arp                   1 192.150.187.37
arp r                 : 0:c0:4f:68:8c:58
192.1                 .87.20.ssh: S [27469] 23916613:23916613(0) win 16384 <mss 1460> (DF)
192.1                 .87.37.938: S [28831] 3641908505:3641908505(0) ack 23916614 win 17520
192.1                 .87.20.ssh: . [34908] ack 1 win 17520 (DF)
192.1                 .87.37.938: P [59094] 1:16(15) ack 1 win 17520 (DF) [tos 0x10]
192.1                 .87.20.ssh: P [41307] 1:23(22) ack 16 win 17520 (DF)
192.1                 .87.37.938: P [21929] 16:292(276) ack 23 win 17520 (DF) [tos 0x10]
192.1                 .87.20.ssh: P [33476] 23:179(156) ack 292 win 17520 (DF)
192.1                 .87.37.938: . [34439] ack 179 win 17520 (DF) [tos 0x10]
192.1                 .87.37.938: P [26772] 292:304(12) ack 179 win 17520 (DF) [tos 0x10]
192.1                 .87.20.ssh: P [18617] 179:207(28) ack 304 win 17520 (DF)
192.1                 .87.37.938: P [4792] 304:364(60) ack 207 win 17520 (DF) [tos 0x10]
192.1                 .87.37.938: P [36223] 364:376(12) ack 207 win 17520 (DF) [tos 0x10]
192.1                 .87.20.ssh: . [34327] ack 376 win 17520 (DF)
192.1                 .87.20.ssh: P [31847] 207:235(28) ack 376 win 17520 (DF)
192.1                 .87.37.938: P [12692] 376:388(12) ack 235 win 17520 (DF) [tos 0x10]
192.150.187.37.938 > 192.150.187.20.ssh: P [17359] 235:383(148) ack 388 win 17520 (DF)
[illegible]
192.150.187.37.938 > 192.150.187.20.ssh: P [49200] 383:395(12) ack 400 win 17520 (DF) [tos 0x10]
192.150.187.20.ssh > 192.150.187.37.938: P [40358] 400:1000(600) ack 395 win 17520 (DF) [tos 0x10]
192.150.187.20.ssh > 192.150.187.37.938: P [47504] 1000:1808(808) ack 395 win 17520 (DF) [tos 0x10]
192.150.187.20.ssh > 192.150.187.37.938: P [43917] 1808:1844(36) ack 395 win 17520 (DF) [tos 0x10]
```

Source Port (22, ssh)								Destination Port (938)
Seq Number (3641908893)								
Ack Number (23918998)								
Hdr Len (20)	Reserved (0)	U	A	P	R	S	F	Window Size (17520)
Checksum (31567)								Urgent Pointer (0)

Figure 5: Using NetDuDE to create test traffic

To this end, we developed NetDuDE (Figure 5), the Network Dump Displayer and Editor. NetDuDE takes `libpcap` packet tracefile, displays the packets graphically, and allows us to examine IP, TCP, UDP, and ICMP header fields.[5] In addition, it allows us to *edit* the tracefile, setting the values of fields, adding and removing options, recalculating checksums, changing the packet ordering, and duplicating, fragmenting, reassembling or deleting packets.

To test a particular normalization, we edit an existing trace to create the appropriate anomalies. We then feed the tracefile through *norm* to create a new normalized trace. We then both reexamine this trace in NetDuDE to manually check that the normalization we intended actually occurred, and feed the trace back into *norm*, to ensure that on a second pass it does not modify the trace further. Finally, we store the input and output tracefiles in our library of anomalous traces so that we can perform automated validation tests whenever we make a change to *norm*, to ensure that changing one normalization does not adversely affect any others.

7.2 Performance

As mentioned above, our current implementation of *norm* runs at user level, but we are primarily interested in assessing how well it might run as a streamlined kernel implementation, since it is reasonable to expect that a production normalizer will merit a highly optimized implementation.

To address this, *norm* incorporates a test mode whereby it reads an entire `libpcap` trace file into memory and in addition allocates sufficient memory to store all the resulting normalized packets. It then times how long it takes to run, reading packets from one pool of memory, normalizing them, and storing the results in the second memory pool. After measuring the performance, *norm* writes the second memory pool out to a `libpcap` trace file, so we can ensure that the test did in fact measure the normalizations we intended.

These measurements thus factor out the cost of getting packets to the normalizer and sending them out once the normalizer is done with them. For a user-level implementation, this cost is high, as it involves copying the entire packet stream up from kernel space to user space and then back down again; for a kernel implementation, it should be low (and we give evidence below that it is).

For baseline testing, we use three tracefiles:

Trace T1: a 100,000 packet trace captured from the Internet access link at the Lawrence Berkeley National Laboratory, containing mostly TCP traffic (88%) with some UDP (10%), ICMP (1.5%), and miscellaneous (IGMP, ESP, tunneled IP, PIM; 0.2%). The mean packet size is 491 bytes.

Trace U1: a trace derived from T1, where each TCP header has been replaced with a UDP header. The IP parts of the packets are unchanged from T1.

Trace U2: a 100,000 packet trace consisting entirely of 92 byte UDP packets, generated using *netcat*.

[5] At the time of writing, ICMP support is still incomplete.

T1 gives us results for a realistic mix of traffic; there's nothing particularly unusual about this trace compared to the other captured network traces we've tested. U1 is totally unrealistic, but as UDP normalization is completely stateless with very few checks, it gives us a baseline number for how expensive the more streamlined IP normalization is, as opposed to TCP normalization, which includes many more checks and involves maintaining a control block for each flow. Trace U2 is for comparison with U1, allowing us to test what fraction of the processing cost is per-packet as opposed to per-byte.

We performed all of our measurements on an x86 PC running FreeBSD 4.2, with a 1.1GHz AMD Athlon Thunderbird processor and 133MHz SDRAM. In a barebones configuration suitable for a normalizer box, such a machine costs under US$1,000.

For an initial baseline comparison, we examine how fast *norm* can take packets from one memory pool and copy them to the other, without examining the packets at all:

Memory-to-memory copy only		
Trace	pkts/sec	bit rate
T1,U1	727,270	2856 Mb/s
U2	1,015,600	747 Mb/s

Enabling all the checks that *norm* can perform for both inbound and outbound traffic[6] examines the cost of performing the tests for the checks, even though most of them entail no actual packet transformation, since (as in normal operation) most fields do not require normalization:

All checks enabled		
Trace	pkts/sec	bit rate
T1	101,000	397 Mb/s
U1	378,000	1484 Mb/s
U2	626,400	461 Mb/s

Number of Normalizations					
Trace	IP	TCP	UDP	ICMP	Total
T1	111,551	757	0	0	112,308

[6] Normally fewer checks would be enabled for outbound traffic.

Comparing against the baseline tests, we see that IP normalization is about half the speed of simply copying the packets. The large number of IP normalizations consist mostly of simple actions such as TTL restoration, and clearing the DF and Diffserv fields. We also see that TCP normalization, despite holding state, is not vastly more expensive, such that TCP/IP normalization is roughly one quarter of the speed of UDP/IP normalization.

These results do not, of course, mean that a kernel implementation forwarding between interfaces will achieve these speeds. However, the Linux implementation of the *click* modular router [7] can forward 333,000 small packets/sec on a 700MHz Pentium-III. The results above indicate that normalization is cheap enough that a normalizer implemented as (say) a *click* module should be able to forward normal traffic at line-speed on a bi-directional 100Mb/s link.

Furthermore, if the normalizer's incoming link is attacked by flooding with small packets, we should still have enough performance to sustain the outgoing link at full capacity. Thus we conclude that deployment of the normalizer would not worsen any denial-of-service attack based on link flooding.

A more stressful attack would be to flood the normalizer with small fragmented packets, especially if the attacker generates out-of-order fragments and intersperses many fragmented packets. Whilst a normalizer under attack can perform triage, preferentially dropping fragmented packets, we prefer to only do this as a last resort.

To test this attack, we took the T1 trace and fragmented every packet with an IP payload larger than 16 bytes: trace T1-frag comprises some 3 million IP fragments with a mean size of 35.7 bytes. Randomizing the order of the fragment stream over increasing intervals demonstrates the additional work the normalizer must perform. For example, with minimal re-ordering the normalizer can reassemble fragments at a rate of about 90Mb/s. However, if we randomize the order of fragments by up to 2,000 packets, then the number of packets simultaneously in the fragmentation cache grows to 335 and the data rate we can handle halves.

rnd intv'l	input frags/s	frag'ed bit rate	output pkts/sec	output bit rate	pkts in cache
100	299,670	86Mb/s	9,989	39Mb/s	70
500	245,640	70Mb/s	8,188	32Mb/s	133
1,000	202,200	58Mb/s	6,740	26Mb/s	211
2,000	144,870	41Mb/s	4,829	19Mb/s	335

It is clear that in the worst case, *norm* does need to perform triage, but that it can delay doing so until a large fraction of the packets are very badly fragmented, which is unlikely except when under attack.

The other attack that slows the normalizer noticeably is when *norm* has to cope with inconsistent TCP retransmissions. If we duplicate every TCP packet in T1, then this stresses the consistency mechanism:

All checks enabled		
Trace	pkts/sec	bit rate
T1	101,000	397 Mb/s
T1-dup	60,220	236 Mb/s

Although the throughput decreases somewhat, the reduction in performance is not grave.

To conclude, a software implementation of a traffic normalizer appears to be capable of applying a large number of normalizations at line speed in a bi-directional 100Mb/s environment using commodity PC hardware. Such a normalizer is robust to denial-of-service attacks, although in the specific case of fragment reassembly, very severe attacks may require the normalizer to perform triage on the attack traffic.

Acknowledgments

We'd like to thank Bill Fenner, Brad Karp, Orion Hodson, Yin Zhang, Kevin Fall, Steve Bellovin and the End-to-end Research Group for their comments and suggestions. Thanks also go to Jupiter's in Berkeley and the fine Guinness and Lost Coast brewing companies for lubricating many hours of discussion on this topic.

References

[1] M. Allman, D. Glover and L. Sanchez, "Enhancing TCP Over Satellite Channels using Standard Mechanisms," RFC 2488, Jan. 1999.

[2] Anzen Computing, *fragrouter*, 1999. http://www.anzen.com/research/nidsbench/

[3] Fyodor, *nmap*, 2001. http://www.insecure.org/nmap/

[4] M. Handley, C. Kreibich, and V. Paxson, draft technical report, to appear at http://www.aciri.org/vern/papers/norm-TR-2001.ps.gz, 2001.

[5] horizon `<jmcdonal@unf.edu>`, "Defeating Sniffers and Intrusion Detection Systems", Phrack Magazine Volume 8, Issue 54, Dec. 25th, 1998.

[6] C. Kent and J. Mogul, "Fragmentation Considered Harmful," *Proc. ACM SIGCOMM*, 1987.

[7] E. Kohler, R. Morris, B. Chen, J. Jannotti and M.F. Kaashoek, "The Click modular router," *ACM Transactions on Computer Systems*, 18(3), pp. 263–297, Aug. 2000.

[8] G. R. Malan, D. Watson, F. Jahanian and P. Howell, "Transport and Application Protocol Scrubbing", Proceedings of the IEEE INFOCOM 2000 Conference, Tel Aviv, Israel, Mar. 2000.

[9] L. Deri and S. Suin, "Improving Network Security Using Ntop," *Proc. Third International Workshop on the Recent Advances in Intrusion Detection (RAID 2000)*, Toulouse, France, Oct. 2000.

[10] S. McCanne, C. Leres and V. Jacobson, `libpcap`, 1994. ftp://ftp.ee.lbl.gov/libpcap.tar.Z

[11] K. Nichols, S. Blake, F. Baker and D. Black, "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers", RFC 2474, Dec. 1998.

[12] V. Paxson, "Bro: A System for Detecting Network Intruders in Real-Time", Computer Networks, 31(23-24), pp. 2435-2463, 14 Dec 1999.

[13] V. Paxson and M. Handley, "Defending Against NIDS Evasion using Traffic Normalizers," presented at *Second International Workshop on the Recent Advances in Intrusion Detection*, Sept. 1999.

[14] T. H. Ptacek and T. N. Newsham, "Insertion, Evasion and Denial of Service: Eluding Network Intrusion Detection", Secure Networks, Inc., Jan. 1998. http://www.aciri.org/vern/Ptacek-Newsham-Evasion-98.ps

[15] K. Ramakrishnan and S. Floyd, "A Proposal to add Explicit Congestion Notification (ECN) to IP", RFC 2481, Jan. 1999.

[16] S. Sanfilippo, "new tcp scan method," *Bugtraq*, Dec. 18, 1998.

[17] M. Smart, G.R. Malan and F. Jahanian, "Defeating TCP/IP Stack Fingerprinting," *Proc. USENIX Security Symposium*, Aug. 2000.

[18] M. de Vivo, E. Carrasco, G. Isern and G. de Vivo, "A Review of Port Scanning Techniques," *Computer Communication Review*, 29(2), April 1999.

A Normalizations performed by *norm*

Our normalizer implementation *norm* currently performs 54 of the following 73 normalizations we identified:

IP Normalizations

#	IP Field	Normalization Performed
1	Version	Non-IPv4 packets dropped.
2	Header Len	Drop if hdr_len too small.
3	Header Len	Drop if hdr_len too large.
4	Diffserv	Clear field.
5	ECT	Clear field.
6	Total Len	Drop if tot_len > link layer len.
7	Total Len	Trim if tot_len < link layer len.
8	IP Identifier	Encrypt ID.†
9	Protocol	Enforce specific protocols.†
–	Protocol	Pass packet to TCP,UDP,ICMP handlers.
10	Frag offset	Reassemble fragmented packets.
11	Frag offset	Drop if offset + len > 64KB.
12	DF	Clear DF.
13	DF	Drop if DF set and offset > 0.
14	Zero flag	Clear.
15	Src addr	Drop if class D or E.
16	Src addr	Drop if MSByte=127 or 0.
17	Src addr	Drop if 255.255.255.255.
18	Dst addr	Drop if class E.
19	Dst addr	Drop if MSByte=127 or 0.
20	Dst addr	Drop if 255.255.255.255.
21	TTL	Raise TTL to configured value.
22	Checksum	Verify, drop if incorrect.
23	IP options	Remove IP options.†
24	IP options	Zero padding bytes.†

† Indicates normalizations planned, but either not yet implemented or not yet tested at the time of writing.

Note that most normalizations are optional, according to local site policy.

UDP Normalizations

#	UDP Field	Normalization Performed
1	Length	Drop if doesn't match length as indicated by IP total length.
2	Checksum	Verify, drop if incorrect.

TCP Normalizations

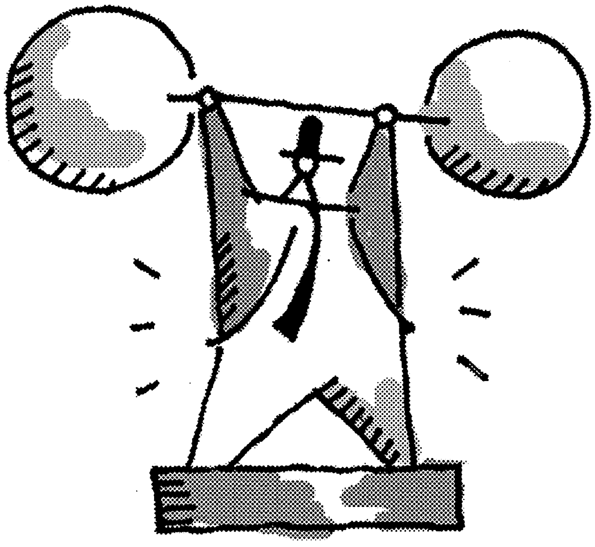
#	TCP Field	Normalization Performed
1	Seq Num	Enforce data consistency in retransmitted segments.
2	Seq Num	Trim data to window.
3	Seq Num	Cold-start: trim to keep-alive.
4	Ack Num	Drop ACK above sequence hole.
5	SYN	Remove data if SYN=1.
6	SYN	If SYN=1 & RST=1, drop.
7	SYN	If SYN=1 & FIN=1, clear FIN.
8	SYN	If SYN=0 & ACK=0 & RST=0, drop.
9	RST	Remove data if RST=1.
10	RST	Make RST reliable.
11	RST	Drop if not in window.†
12	FIN	If FIN=1 & ACK=0, drop.
13	PUSH	If PUSH=1 & ACK=0, drop.
14	Header Len	Drop if less than 5.
15	Header Len	Drop if beyond end of packet.
16	Reserved	Clear.
17	ECE, CWR	Optionally clear.
18	ECE, CWR	Clear if not negotiated.†
19	Window	Remove window withdrawals.
20	Checksum	Verify, drop if incorrect.
21	URG,urgent	Zero urgent if URG not set.
22	URG,urgent	Zero if urgent > end of packet.
23	URG	If URG=1 & ACK=0, drop.
24	MSS option	If SYN=0, remove option.
25	MSS option	Cache option, trim data to MSS.
26	WS option	If SYN=0, remove option.
27	SACK pmt'd	If SYN=0, remove option.
28	SACK opt	Remove option if length invalid.†
29	SACK opt	Remove if left edge of SACK block > right edge.†
30	SACK opt	Remove if any block above highest seq. seen.†
31	SACK opt	Trim any block(s) overlapping or continguous to cumulative acknowledgement point.†
32	T/TCP opts	Remove if NIDS doesn't support.
33	T/TCP opts	Remove if under attack.†
34	TS option	Remove from non-SYN if not negotiated in SYN.†
35	TS option	If packet fails PAWS test, drop.†
36	TS option	If echoed timestamp wasn't previously sent, drop.†
37	MD5 option	If MD5 used in SYN, drop non-SYN packets without it.†
38	*other opts*	Remove options.

ICMP Normalizations

#	ICMP Type	Normalization Performed
1	Echo request	Drop if destination is a multicast or broadcast address.
2	Echo request	Optionally drop if ping checksum incorrect.
3	Echo request	Zero "code" field.
4	Echo reply	Optionally drop if ping checksum incorrect.
5	Echo reply	Drop if no matching request.†
6	Echo reply	Zero "code" field.
7	Source quench	Optionally drop to prevent DoS.†
8	Destination Unreachable	Unscramble embedded scrambled IP identifier.†
9	*other*	Drop.†

The following "transport" protocols are recognized, but currently passed through unnormalized: IGMP, IP-in-IP, RSVP, IGRP, PGM.

OPERATING SYSTEMS



Session Chair: Teresa Lunt, *Xerox PARC*

Security Analysis of the Palm Operating System and its Weaknesses Against Malicious Code Threats

Kingpin and Mudge

@stake, Inc.
196 Broadway
Cambridge, MA 02139
{kingpin,mudge}@atstake.com

Abstract

Portable devices, such as Personal Digital Assistants (PDAs), are particularly vulnerable to malicious code threats due to their widespread implementation and current lack of a security framework. Although well known in the security industry to be insecure, PDAs are ubiquitous in enterprise environments and are being used for such applications as one-time-password generation, storage of medical and company confidential information, and e-commerce. It is not enough to assume all users are conscious of computer security and it is crucial to understand the risks of using portable devices in a security infrastructure. Furthermore, it is not possible to employ a secure application on top of an insecure foundation.

Palm operating system (OS) devices own nearly 80 percent of the global handheld computing market [11]. It is because of this that the design of the Palm OS and its supporting hardware platform were analyzed. The presented research provides detail into specific scenarios, weaknesses, and mitigation recommendations related to data protection, malicious code, virus storage, and virus propagation. Additionally, this work can be used as a model by users and developers to gain a deeper understanding of the additional security risks that these and other portable devices introduce.

*Palm OS and HotSync are registered trademarks of Palm, Inc. Other product and company names may be trademarks of their respective owners.

1 Introduction

A new threat model exists for malicious code and virus attacks on portable devices. These threats are no longer contained to common desktop environments. Portable devices employing custom electrical circuit design, product-specific capabilities, and embedded operating systems are commonplace in corporate infrastructure. It is increasingly common for vendors to introduce these devices to an environment before the security ramifications have been examined. PDAs are now being deployed by corporations for security-related applications. Added functionality of wireless technologies, such as infrared (IR) and radio frequency (RF), increases risk areas. New classes of malicious code attacks exist that cannot be detected or contained by current methods long deployed in desktop environments. In addition, the notion of cross-architecture pollination very quickly becomes a mainstream concern. [5] provides an overview of some malicious threats to PDAs and can be read in parallel with this text.

Many users do not recognize that the information stored on their PDA is open to compromise by unauthorized users, and hence do not treat the data stored on their handhelds with the same care as they do on their desktop. Our research discusses the underlying problem that security is not properly designed into the Palm OS platform. Although Palm OS is not presented as a secure operating system, if the device is being used for security purposes, which is becoming prevalent in corporate environments, there are a number of risk areas to be concerned with.

For example, Palm OS offers a built-in Security application which is used for the legitimate user to protect and hide records from unauthorized users by means of a password. In all basic built-in applications (Address, Date Book, Memo Pad, and To Do List), individual records can be marked as "Private" and should only be accessible if the correct password is entered. Another example is the "Beam Bit" flag contained in every application database, which is used to prevent the information from being transferred, or "beamed", to another device via IR. Honoring the state of the Beam Bit is purely voluntary by the executing application. These simplistic mechanisms lull the user and perhaps some developers into a false sense of security. There should be strong warnings by the vendor that these mechanisms are trivially bypassed (as in §4, §5, and with [14]), so users and developers can plan for and workaround the lack of security. Security-based applications exist on the Palm OS, such as software authentication tokens, cryptographic key storage, and encryption products, all that require a secure operating system in order to be properly implemented. Without proper protection mechanisms in place, applications that rely on the secure storage of secret components are severely at risk of compromise.

The properties of malicious code, particularly viruses, can be distilled into four stages: Infection, Storage, Triggers, and Actions. In this paper, the design of Palm OS is analyzed with respect to each of these stages. A number of weaknesses and attack vectors have been identified from both classical and new technology areas and we offer insight into addressing these problems in design and usage. In no way is this text exhaustive in enumerating attacks. Rather, an attempt is made to educate the reader on the design flaws and new threats that exist on portable devices.

In §2, we provide a summary of the various types of malicious code: viruses, Trojan horses, and worms. §3 describes the typical design and architecture of a PDA, focusing on the Palm OS software and hardware platform. §4 and §5 detail the risks of weak system password storage and backdoor debug modes inherent in Palm OS. §6 through §9 address the four stages of the virus lifecycle with respect to Palm OS.

We conclude that current state-of-the-art portable devices are not equipped for the threat of viruses or other malicious code components. In addition, it becomes apparent that threat models and attack vectors these devices introduce are not yet taken into account by product designers and anti-virus vendors[1]. Hopefully, the various sections of this paper can act as a road map towards the future design of these devices and aid in security awareness for existing deployments.

2 Summary of Malicious Code Types

For the purposes of clarity, we will classify malicious code into three areas [23]:

- **A Virus** is a self-replicating code segment which must be attached to a host executable. When the host is executed, the virus code may also execute. If possible, the virus will replicate by attaching a copy of itself to another executable. The virus may include an additional "payload" that triggers when specific conditions are met.
- **A Trojan horse** is malicious code masquerading as a legitimate application. The goal of the code is to have the user believe they are conducting standard operations or running an innocuous application when in fact initiating its ulterior activities. There are many ways this attack manifests with the most frequent being reliance upon user naivety. A Trojan horse is similar to a virus, except a Trojan horse does not replicate.
- **A Worm** is a self-replicating program. It is self-contained and does not require a host program. The program creates the copy and causes it to execute; no user intervention is required. Worms commonly utilize network services to propagate to other computer systems [19].

3 Palm OS Device Architecture

At the highest level, the architecture of the Palm OS device, and most other PDAs, can be broken down into three layers (Figure 1): Application, Operating System, and Hardware.

Use of the Palm OS Application Programming Interface (API) provides the application developer

[1]Anti-virus software for PDAs is available from a number of vendors, including, but not limited to: Central Command, F-Secure, McAfee.com, Symantec, and Trend Micro.

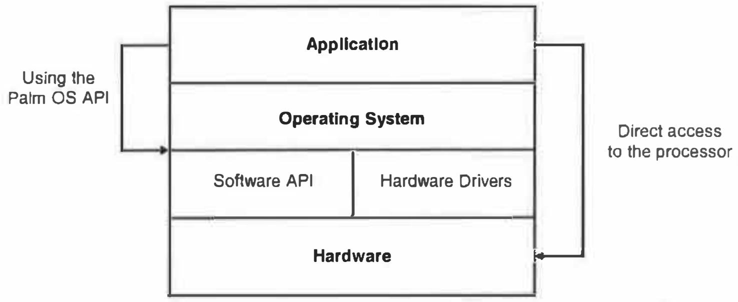


Figure 1: Typical layered architecture of a PDA

with a notion of hardware independence and provides a layer of abstraction. If the API is used properly, recompiling of the application is all that is necessary in order to run on Palm OS devices based on different hardware. Therefore, it is important to examine weaknesses and attack vectors that can be found at the programming interface to the operating system.

Directly accessing the processor by avoiding the interface put forward by the operating system allows the developer to have more control of the processor and its functionality. A risk of legitimate use of direct processor access is the loss of compatibility for future models. For example, older Palm OS devices did not support a grayscale LCD palette through the Palm OS API, even though the underlying hardware possessed this capability. Bypassing this interface and tapping into the functionality of the processor directly will remedy this [13]. Ideally, to provide some semblance of access control and security, only the operating system should have access to the underlying hardware. Allowing applications to directly access hardware provides an avenue for malicious attack (as discussed in §9.2).

3.1 Operating System

Palm OS was designed to be open and modular to support application development by third-parties. The notion of layer- or file-based access control is notably absent. It is not surprising that all program code and data can be accessed and modified by any user or other application. In such uniform memory access scenarios, it is difficult to differentiate between legitimate and malicious applications solely from memory read/writes and system calls.

[20] offers the following overview on file system and application structure:

- Palm OS does not use a traditional flat file system. Data is stored in memory chunks called "records", which are grouped into "databases". A database is analogous to a file. The difference is that data is broken down into multiple records instead of being stored in one contiguous chunk.
- Palm OS applications are generally single-threaded, event-driven programs. Only one program runs at a time. Each application has a `PilotMain` function that is equivalent to `main` in C programs. To launch an application, Palm OS calls `PilotMain` and sends it a launch code. The launch code may specify that the application is to become active and display its user interface (called a "normal launch"), or it may specify that the application should simply perform a small task and exit without displaying its user interface. The sole purpose of the `PilotMain` function is to receive launch codes and respond to them. Future versions of the Palm OS may allow third-party applications to be multi-threaded.
- Applications can send launch codes to each other, so an application might be launched from another application or it might be launched from the system. An application can use a launch code to request that another application perform an action or modify its data.

3.2 Hardware

All Palm OS devices, including those by Handspring, Sony, IBM, Kyocera, QUALCOMM, Franklin Covey, TRG and Symbol Technologies, currently use the Motorola DragonBall MC68328-family of microprocessors which are based on the Motorola MC68EC000 core[2]. The DragonBall processors are inherently low-speed, ranging from 16MHz to 33MHz depending on the type (MC68328, 'EZ328, or 'VZ328). ARM Limited's microprocessor architecture, employed in many consumer, wireless, and security products, will be used as the core of future DragonBall processors [2] and is planned to be implemented in Palm OS devices in 2002.

Palm OS and other handheld embedded devices use battery-backed Random Access Memory (RAM) to store application and user data. The operating system and other non-transient components are often stored in Read-Only Memory (ROM). However, newer devices are moving towards Flash memory for static components such as the operating system. Flash memory is non-volatile and the data stored in it will remain intact even with loss of battery power or a hard reset. The Palm OS is restarted from its ROM or Flash storage area upon system reset.

4 Retrieval of Passwords

It is possible, via a number of methods, to extract data from portable devices by reading raw memory or from the host system after such data has been backed up. These attacks can retrieve files containing potentially valuable data such as passwords, financial, medical, or other company or personal information. In officially sanctioned scans, the authors found that the passwords chosen by users to protect data on their PDAs were the same as those being used for critical corporate assets.

One example of a high-security application is medical data, which is increasingly being stored on portable devices by doctors in order to have immediate access to patient information. Recent situations have occurred in which hospital intruders have beamed extensive amounts of unprotected patient data off of Palm OS devices. This could have been avoided with the proper use of passwords, encryption, and access-control on the device.

[2]Motorola's *MC68328 DragonBall Integrated Processor User's Manual* describes the programming, capabilities, and operation of the MC68328; the *M68000 Microprocessor User's Manual* provides instruction details for the 'EC000 core.

History has shown the weaknesses of poorly chosen or stored passwords, as in [17] and with the Morris Worm [19]. Users of portable devices, especially those that have no keyboard and require character input with a pen, oftentimes choose short, easily guessable passwords, placing convenience over security. Leveraging this, the scenario presents itself where malicious code determines the user's password on the local device and, upon connection to a network or other system, attempts to gain access to other systems using the user name and now-known password. This type of attack ends up being disconcertingly successful.

As it happens, an encoded block is stored on the Palm OS device in the Unsaved Preferences database that contains a reversible obfuscation of the user's system password [15]. The block is not only readable by any application on the actual device, but is also transmitted over the serial cable, airwaves, and networks during a HotSync operation. This problem is verified to concern Palm OS versions 3.5.2 and earlier.

4.1 Password Decoding Details

The password is set by the legitimate user with the Security application. The maximum length of the ASCII password is 31 characters. Regardless of the length of the ASCII password, the resultant encoded block is always 32 bytes. Two methods are used to encode the ASCII password, depending on its length. For passwords of four characters or fewer, an index is calculated based on the length of the password and the string is XORed against a 32-byte constant block. For passwords of more than four characters, the string is padded to 32 bytes and run through four rounds of a function that XORs against a 64-byte constant block. By understanding the encoding schema, it is possible to essentially run the routines in reverse to decode the password.

The Palm desktop software makes use of the Serial Link Protocol (SLP) to transfer information between itself and the Palm device. Each SLP packet consists of a packet header, client data of variable size, and a packet footer [20]. During the HotSync negotiation process, one particular SLP packet's client data consists of a structure which contains the encoded password block (Figure 2).

```
struct {
  UInt8 header[4];
  UInt8 exec_buf[6];
  Int32 userID;
  Int32 viewerID;
  Int32 lastSyncPC;
  time_t successfulSyncDate;
  time_t lastSyncDate;
  UInt8 userLen;
  UInt8 passwordLen;
  UInt8 username[userLen+1];
  UInt8 password[passwordLen+1];
};
```

Figure 2: Structure sent during the HotSync process containing encoded password block

Passwords of 4 Characters or Less: By comparing the encoded password blocks of various short passwords (example in Figure 3), it was determined that a 32-byte constant (Figure 4) was simply being XORed against the ASCII password block.

A = ASCII password
B = 32-byte constant block
C = encoded password block

The starting index, j, into the constant block where the XOR operation should begin is calculated by:

```
j = (A[0] + strlen(A)) % 32;
```

The encoded password block is then created:

```
for (i = 0; i < 32; ++i, ++j)
{
   // wrap around to beginning
   if (j == 32) j = 0;

   C[i] = A[i] XOR B[j];
}
```

```
56 8C D2 3E 99 4B 0F 88 09 02 13 45 07 04 13 44
0C 08 13 5A 32 15 13 5D D2 17 EA D3 B5 DF 55 63
```

Figure 3: Encoded password block of ASCII password 'test'

Passwords Greater Than 4 Characters: The encoding scheme for long length passwords (up to 31 characters in length) is more complicated than for short length passwords, although it, too, is reversible.

```
09 02 13 45 07 04 13 44 0C 08 13 5A 32 15 13 5D
D2 17 EA D3 B5 DF 55 63 22 E9 A1 4A 99 4B 0F 88
```

Figure 4: 32-byte constant block for use with passwords of length 4 characters or less

A = ASCII password
B = 64-byte constant block
C = encoded password block

First, A is padded to 32 bytes in the following fashion:

```
j = strlen(A);

while (j < 32)
{
   for (i = j; i < j * 2; ++i)
      // increment each ASCII value by j
      A[i] = A[i - j] + j;

   j = j * 2;
}
```

The resultant 32-byte array, A, is then passed through four rounds of a function which XORs against a 64-byte constant (Figure 5). k is an index that begins at {2,16,24,8} for each of the four rounds.

```
j = (A[k] + A[k+1]) & 0x3F; // 6 LSB
shift = (A[k+2] + A[k+3]) & 0x07; // 3 LSB

for (i = 0; i < 32; ++i, ++j, ++k)
{
   // wrap around to beginning
   if (j == 64) j = 0;
   if (k == 32) k = 0;

   temp = B[j]; // xy
   temp <<= 8;
   temp |= B[j]; // xyxy

   temp >>= shift;

   C[k] XOR= (unsigned char) temp;
}
```

The resultant 32-byte encoded password block (example in Figure 6) does not have any immediately visible remnants of the constant block as the short length encoding method does. However, it is still reversible with minimal computing resources.

```
B1 56 35 1A 9C 98 80 84 37 A7 3D 61 7F 2E E8 76
2A F2 A5 84 07 C7 EC 27 6F 7D 04 CD 52 1E CD 5B
B3 29 76 66 D9 5E 4B CA 63 72 6F D2 FD 25 E6 7B
C5 66 B3 D3 45 9A AF DA 29 86 22 6E B8 03 62 BC
```

Figure 5: 64-byte constant block for use with passwords greater than 4 characters

```
18 0A 43 3A 17 7D A3 CA D7 9D 75 D2 D3 C8 A5 CF
F1 71 07 03 5A 52 4B B9 70 2D B2 D1 DF A5 54 07
```

Figure 6: Encoded password block of ASCII password 'testa'

4.2 Recommendations

Palm OS 4.0, due to be released at the end of 2001, appears to have resolved the issue of weak password obfuscation. However, it is highly recommended that a thorough analysis of OS 4.0 takes place before a security-critical application is deployed.

In the current state, it is recommended that Palm OS devices should not be trusted to store any critical or confidential information. In lieu of this, users and vendors are encouraged to adhere to the following guidelines for increased password security:

- **Engage a challenge/response mechanism.** These mechanisms will minimize the potential for adversaries to glean passwords through passive monitoring of the transport medium. The transfer of a secret component, even if it is encoded or obfuscated, over accessible buses (e.g., serial, IR, wireless, or network) is a risky design decision. Unfortunately, it's common practice that applications choose to simply obfuscate passwords instead of using encryption.
- **Encrypt and salt credentials stored on systems.** Simple obfuscation and reversible transforms lull the user into a false sense of security and simultaneously show a lack of concern about security from the vendor. The use of a salt, such as the Palm user name, user ID, or unique serial number of the Palm device, minimizes the possibilities of a password being represented on multiple systems with the same hash.
- **Implement policy to lock and encrypt data on the device.** The Palm OS Security application provides "system lockout" functionality in which the Palm device will not be operational until the correct password is entered. This is meant to prevent an unauthorized user from reading data or running applications on the device. Although this protection can be bypassed as discussed in §5, it provides an additional layer of security for particular deployments. The encryption of data can be achieved with a number of third-party applications, though care should be taken to verify secure storage of the encryption components.
- **Implement an alternative password scheme.** Third-party solutions exist which provide power-on and data protection by requiring a handwritten signature, physical button taps, or other form of password before allowing access to the device. Ths use of graphical passwords on PDAs is studied in [12].

5 Backdoor Debug Modes

Designed into the Palm OS is an RS232-based "Palm Debugger", which provides source- and assembly-level debugging of Palm OS executables and the administration of databases existing on the physical device [21].

Entering a short keystroke combination [21], the Palm OS device enters one of two interfaces provided by the Palm Debugger and monitors the serial port for communication. "Console mode" interacts with a high-level debugger and is used mostly for the manipulation of databases. "Debug mode" is typically used for assembly- and register-level debugging. A soft-reset of the Palm device will exit debug mode, leaving no proof of prior use.

The Palm Debugger can be activated even if the Palm OS lockout functionality is enabled (which is currently assumed by most users to be a sufficient protection feature, because a password is required before the device becomes operational). This problem is verified to concern Palm OS versions 3.5.2 and earlier.

Aside from the specific attack of retrieving the obfuscated system password block by using `export 0 "Unsaved Preferences"` and decoding as detailed in §4.1, it is possible to access all database and record information on the entire Palm OS device [16]. For example, using the `import` console command, one can load a Palm OS application into the device, therefore side-stepping any HotSync or beaming operations and logging mechanisms. A complete listing of console and debug commands can be found in [21].

Because the debug modes communicate with the host via the serial port, it would be possible to create a Palm OS-based application to emulate the required commands and, with a modified HotSync cable, be used for the retrieval of passwords or other data in a mobile fashion. When the possibility exists to retrieve data from a portable device while "in the field" and not requiring the use of a desktop computer, the threat of physical attacks increases greatly.

5.1 Recommendations

Solutions for this class of attack can be remedied with minimal changes to the Palm OS. If the device has been placed in the system lockout mode, the Palm Debugger functionality should be disabled. Palm OS 4.0 appears to have removed the activation of debug functionality during the "system lockout" mode. In an ideal situation, although a disadvantage to application developers, all debugging functionality should be removed in production devices.

Additionally, logging all Palm Debugger actions, especially with time stamping, aims towards forensics readiness and will aid in post-attack analysis.

If access control features are implemented in future Palm OS versions, as they should be, it should be noted that the permissions remain intact during debug sessions and that global memory accessibility is not allowed.

6 Infection Techniques

Common to most virus applications, and intrinsic to worms, is the notion of self-replication. Through self-replication and propagation, the malignant code can infect programs, devices, users, or combinations thereof. Hence, it is important to look at avenues available to such programs to better understand the risks at hand and determine areas to analyze for solutions.

Generic applications can be loaded in a number of different fashions. They can even execute without user knowledge or interaction. Any method of loading data onto the Palm OS device can act as an entry point for virus or malicious code infection. Four major entry points for the Palm OS devices are: HotSync operations, serial ports, infrared beaming, and wireless radio. Additionally, applications can be loaded using the Palm Debugger as described in §5.

Possibly more threatening and intriguing is the potential for cross-architecture pollination and infection. As with biology, the life cycle of a pathogen may involve more than one species of host. A virus could easily be designed to infect a desktop PC and contain a secondary payload for the Palm OS device. Alternatively, a virus on a Palm OS device could contain a payload aimed to compromise a desktop PC.

6.1 Application Installation Procedure

The current installation procedure for loading third-party applications onto a Palm OS device is simplistic in nature and was not designed with security in mind. The Install Tool, provided with the Palm Desktop software, copies the desired application into the `/Palm/<user>/Install` directory on the desktop PC. Upon the next HotSync operation, the contents in this directory are automatically loaded onto the Palm OS device. This is one example of cross-architecture pollination as the virus effectively transfers itself to the new platform.

No confirmation or authentication mechanisms exist during the HotSync operation. This shows the integrity and security of the host PC as an integral component in this chain of actions. If the host PC is compromised, the PDA can be considered compromised, as well.

6.1.1 Recommendations

Since the user places each individual program in the directory or otherwise intentionally labels the applications to be uploaded, user verification at synchronization to confirm the applications should be a trivial solution. This could be achieved by automated prompting on the host PC or by manually inspecting the contents of the `/Palm/<user>/Install` directory. However, many users have a learned behavior to simply accept system prompts without careful examination.

Cryptographic signing of applications by the vendor then verified by the user or Palm device will also reduce the chances of illegitimate code being loaded or executed on the device.

6.2 Desktop Conduits

"Conduits", in the form of Dynamic Link Libraries (DLLs), interface with the HotSync Manager program on the desktop PC. They enable the transfer of data between the Palm OS device and a specific desktop application during the HotSync process.

The standard conduits for Palm OS transfer Address, Date Book, Memo Pad, and To Do List data to the Palm Desktop software. Palm Expense data interfaces directly with Microsoft Excel. Third-party conduits exist which replace the standard conduits and will route data to Microsoft Outlook or Exchange, Lotus Notes, Novell GroupWise, or other Personal Information Manager (PIM).

Conduits are an extremely likely entry point for the cross-architecture transfer of malicious code. Aside from virus infection (such as a macro virus through the use of Microsoft Word or Excel macro functionality), malicious code transferred from the Palm device to the desktop through a conduit could exploit a known security problem in the destination desktop application. This could lead to compromise of the desktop machine (such as the execution of arbitrary code, theft or erasure of data, or elevation of privilege).

6.2.1 Recommendations

Cross-architecture infection risks exist for any portable device that employs data transfer or synchronization capabilities to other devices. Proper security practices should exist in the desktop environment consisting of, but not limited to, disabling macros, scripting, and the unprompted execution of code. Anti-virus software running on the desktop should scan the incoming data before passing it to the destination application. Once the malicious code has successfully been transferred to the destination application, it poses the same threats as if a user executed such a file directly.

6.3 Creator ID Replacement

Applications running on the Palm OS make use of a 4-byte Creator ID for identification purposes. If the Creator ID of a malicious application is defined to be the same as one of the built-in applications, it will be executed in place of the built-in application. Launching a Trojan program in this manner will appear transparent to the user until it is too late and the malicious action has occurred. Creator IDs of the basic built-in applications are listed in Table 1.

This behavior has characteristics of a list created in a Last In First Out (LIFO) fashion. Upon addition of a new piece of software to the system, its Creator ID is pushed onto the list. When a program is launched, a traversal of the list occurs to find the entry point to the program. When the first match on the Creator ID is found, the list traversal exits.

Application Name	Creator ID
Address	`addr`
Calculator	`calc`
Date Book	`date`
Expense	`exps`
HotSync	`sync`
Mail	`mail`
Memo Pad	`memo`
Preferences	`pref`
Security	`secr`
To Do List	`todo`

Table 1: Creator IDs of the basic Palm OS built-in applications

6.3.1 Recommendations

Vendors can prevent this problem by monitoring the Creator IDs at the operating system layer and disallowing duplicates. Furthermore, a complete traversal of the list could take place upon each application launch and if duplicate Creator IDs are found, neither application is executed and user intervention would be required. While this opens a window for denial-of-service-style attacks, it closes an obvious Trojan horse attack which is potentially much more damaging.

6.4 Wireless Communications

6.4.1 Infrared

For point-to-point, close quarters communications, infrared is typically the model of choice. In a standard IR beaming session, the Palm OS will send a `sysAppLaunchCmdExgAskUser` launch code to the receiving application. Typically, applications do not have custom handlers for this launch code, in which case the default response is to present the user with a dialog box prompting for acceptance or rejection of the request. If, however, the application handles the launch code, as detailed in §8.1, and sets the `result` flag to `exgAskOk`, the application will send a `sysAppLaunchCmdExgReceiveData` launch code and always receive the incoming data without displaying a dialog box or requiring user intervention.

Using the Exchange Manager functionality in this manner, it is trivial to transmit and receive applications and data over the infrared communications channel. With collusion on the receiving end, as would be possible with an infected system, IR functionality creates a viable conduit for propagation of virus and other malicious applications.

The scenario of beaming business cards at conventions comes quickly to mind as a potential hostile environment that previously might not have been considered as such. Consider a scenario where an adversary, posing as a conference attendee, beams malicious code or other payload, in the form of a business card object, to another individual. The malicious code could then spread from this individual to trusted parties during seemingly innocuous business card transfers.

6.4.2 RF

While infrared beaming is workable in close quarters, other mechanisms must be engaged for wide distance communications. The wireless technology space, particularly RF, has become a primary driver for portable devices. Internet and e-mail connectivity can be obtained through numerous providers, including Novatel Wireless, SkyTel, and Sprint PCS. Wireless Application Protocol (WAP)-capable PDAs and phones are becoming commonplace. Symbol Technologies' family of Palm OS devices integrate a Spectrum24 wireless local-area-network module for enterprise connectivity. The Palm VII employs a radio modem to communicate with the "Palm.Net" service on the Bell South Wireless Data network.

6.4.3 Recommendations

As with any other ingress or egress point on PDAs, wireless technologies create a new vector for possible infection through such means as application transfer or the transmission of intentionally faulty data packets. The design of properly secured wireless networks is beyond the scope of this paper, but it should be noted that if the portable devices are not sufficiently protected, they become a weak link in the transaction process. Consideration should particularly be placed on the storage of secret components (e.g., encryption keys), user authentication, and data transfer mechanisms.

Care should be taken when running server applications on a portable device, particularly when using RF technology (which has a wide operating range). These applications allow other devices to connect inbound to the server device thereby increasing the potential for malicious code to be transferred or for other malicious action (e.g., theft of data) to take place.

Global system functionality that would always prompt for user input and display the applications requested for data reception or transmission would diminish wireless infection. The addition of logging mechanisms for post-mortem analysis would also assist. As these are two suggestions that require vendor intervention, it behooves the user of the device to be cognizant of their surroundings and assess the threat before accepting beamed information from unknown people.

7 Storage and Payload Hiding

A key trait of virus code is the ability to remain invisible to casual scrutiny. This is often accomplished by storing program contents in non-standard areas. While the various methods of encrypting or otherwise obfuscating the payload of a virus program to avoid detection from anti-virus software is beyond the scope of this paper, areas in which code may be attached or stored in Palm OS devices is addressed.

7.1 Preferences and Databases

In the Palm OS API, Preferences and Data Manager functions offer several avenues for data storage. System and application preferences are accessible via the `Pref{Get,Set}Preferences` and `Pref{Get,Set}AppPreferences` function calls. Similarly, any system or application database can be attached to and used to store malicious content. `DmOpenDatabase`, `DmWrite`, `DmResizeRecord`, and `DmSetDatabaseInfo` are all common database manipulation functions that, due to the lack of protection and ownership of individual records, become conduits for attachment.

Unused fields in records are commonly used as covert channels. Databases on the Palm OS device are no exception. For example, the Application and Sort Info Blocks are optional fields in each database that can be used to store application-specific information. Common data stored in this block includes category names or database version numbers. However, it is not necessary for this field to be populated and often times it is not. Traversing the existing database records on the device and checking the `appInfoID` or `sortInfoID` parameter for a `null` pointer will yield a location for the attacker to store the handle (pointer to a location) of their payload. This would not affect the legitimate application's usage in any way.

7.2 Flash Memory

Palm OS devices incorporating non-volatile Flash memory currently use it solely for the storage of the operating system code. Depending on the family of Palm OS device, there remains between 440kB and 824kB of unused memory space.

Utilities exist, such as [27], which make use of the unused memory areas to backup applications and databases. These utilities are OS- and device-specific and use functionality outside of the Palm OS API. This is a perfect example of payload storage and is identical to how a malicious application would utilize Flash memory for such a purpose.

Data could also be stored on the Flash memory outside of the address space that is used by Palm OS, but within the valid memory map as specified in the DragonBall `Group-Base Address` registers. In doing so, applications running on Palm OS using only API functions will not be able to access nor see the data stored in this region.

Recommendations to minimize the risks of improper Flash memory usage are discussed in §9.3.1.

8 Execution Triggers

Viruses do not always execute immediately after infecting a target device. There is often an "incubation period" in which the virus sits dormant, waiting for a specific time, key sequence, or other preordained initiator. The inclusion of an incubation period increases the difficulty of determining exactly how or when the system was infected. As more system activity takes place over time, the ability to backtrack to the point of infection becomes difficult if not impossible.

8.1 Launch Codes

Particular launch codes sent by Palm OS are received by all applications on the Palm device. This becomes a prime candidate for incubation or virus execution, since code segments defined in handling routines are executed without the user's knowledge or intervention. Full details of the launch codes can be found in [22]. A casual perusal of the documentation for launch codes uncovers several obvious events that will likely be used for incubation of malicious code. Our speculations on these are listed in Table 2.

Launch codes are handled in `switch`-style constructs within the `PilotMain` function. An application checks each code that it receives to determine if a handler exists. If one does exist, execution is

Launch Code	Potential Incubation Method
sysAppLaunchCmdSystemReset	This launch code signifies that a system reset has just occurred. No user input is allowed during this launch code. As Palm OS devices are not reset at regular intervals, this provides a random timing for the launch of malicious code.
sysAppLaunchCmdSyncNotify	When a HotSync operation has been completed or an application has been successfully beamed and received by the device, this launch code is sent to application. This could signify that the malicious code has successfully propagated to the target device and can perform its payload hiding or destructive actions.
sysAppLaunchCmdAlarmTriggered	A most probable launch code for malicious use. Malicious code could set an alarm for a future time. Upon receipt of the alarm, the desired code would be executed.

Table 2: Selected application launch codes and theorized incubation methods

handed off to the appropriate functions. The launch code of `sysAppLaunchCmdNormalLaunch`, sent when an application is normally executed, would most often vector to legitimate code. This provides an appearance of normalcy while malicious payloads remain dormant until their specific launch code is seen.

8.1.1 Application Transfer

Through the use of launch codes sent by the Palm OS during the loading of an application (via the HotSync process or IR beaming), it is possible to have an application self-execute after it has been transferred to the target device. Using an infection technique such as described in §6.1, it would be trivial for malicious code to be loaded and executed on a Palm device with the legitimate user having no knowledge of the event.

A typical sequence to execute an application by transfer is as follows:

The newly transferred application will first receive a `sysAppLaunchCmdSyncNotify` launch code from the OS to specify that the device has successfully received the application. If the handling of this launch code sets an alarm for an immediate or future time, the application will be started again with a `sysAppLaunchCmdAlarmTriggered` launch code when that time is reached. The `AppLaunchWithCommand` API function can be called with a `sysAppLaunchCmdNormalLaunch` launch code in order for the application to begin normal execution.

8.1.2 Recommendations

While it is difficult to determine if programs being introduced to the system are malicious in nature, it is possible to sweep existing applications to determine if new launch code handlers have been inserted since the application's original introduction. The modification of an existing program to execute new code at launch would be endemic of viral activity and noticeable through these scans.

8.2 Trap Patching

Well-known to the virus writing community is the notion of "trap patching". When a system function is called, the operating system performs a look-up on the trap dispatch table to determine where in memory the desired function is located. In patching a system function, this address is replaced in the table with an address pointing to new code. Oftentimes, the new code will hand execution off to the original routine after it has served its purpose. In such a scenario, the patch appears invisible to the end user, as the original functionality still succeeds.

Trap patching has many uses beyond that of virus design. For Palm OS devices, trap patching has been made popular with HackMaster [13]. Any native functions in the Palm OS are potential vectors that can be trapped and exploited. This is not only the case for exported user programming interfaces, but includes those that are defined for system-use only.

To help in understanding trap patching as a vulnerability, consider a trivial denial-of-service event:

When a `penUpEvent` event is detected in the writing area, `SysHandleEvent` hands control over to the `GrfProcessStroke` API function. `GrfProcessStroke` is located in the trap dispatch table and the Program Counter starts execution at the address returned. If the `GrfProcessStroke` routine were replaced with a stub that returned immediately after entry, which is to say that the routine does nothing, the attack would result in characters being prevented from entering into the key queue.

Obviously, this constitutes a much more benign attack than ones that might be introduced with greater functionality.

8.2.1 Recommendations

Solutions for this class of problem have been historically difficult [7, 25]. Rollback, in particular, makes the tracking of potentially legitimate patching problematic. For example, take a natural scenario as shown in Figure 7.

$$\begin{tikzcd}
1 \arrow[r] & A \\
2 \arrow[r] & B \\
3 \arrow[r] & C
\end{tikzcd}$$

Figure 7: Functions {1,2,3} with corresponding Addresses {A,B,C}

Assuming that the structure in the trap dispatch table for Function 1 is modified to point to a new Address, D (Figure 8), it would be up to the program that introduced the modification to keep track of the original value.

If yet another patching program is introduced, it would note the native location of Function 1 as Address D. In this case, the second program has no way of knowing that it did not store the original address of Function 1. Upon the first program returning Function 1 to Address A, the second program can still rollback, pushing the return location back to that of Address D.

$$\begin{tikzcd}
 & D \arrow[dr] & \\
1 \arrow[ur] & & A \\
2 \arrow[rr] & & B \\
3 \arrow[rr] & & C
\end{tikzcd}$$

Figure 8: Function 1 patched to point to Address D. Address D hands off to the original location, Address A, upon completion.

Potential exists for periodic checks against vendor-published hash tables to avoid the rollback scenario. It is envisioned that vendors would publish and cryptographically sign a list of the entry points to the various functions. Checks could be made on the portable devices themselves. The Palm OS could also create a list of entry points of newly installed applications and, upon execution, check the stored values against the live values noting discrepancies. A message box or other user alert would be shown should the necessity arise. A cryptographic coprocessor, such as [8, 26], could assist in the secure storage of these entry points.

9 Malicious Actions

9.1 Application Deletion

Without memory protection, it is trivial to create applications capable of deleting program code or database information. The `Palm.Liberty.A` Trojan horse, detected in August 2000 and claimed to be the first known Trojan for the Palm OS platform, did just this in erasing all databases on the device. With complete and unrestricted memory access, the malicious application simply iterates through the linked list of databases and unlinks each one as it proceeds.

9.1.1 Recommendations

There are several preventive approaches for this type of attack. Trapping operating system calls at the API level has been employed in certain scenarios [18]. The calls are often patched to alert the user of a particular action or to disallow an action alto-

Register(s)	Potential Effects

`Phase-Locked Loop (PLL) Control`
`Power Control`
	System can be halted.

`Group-Base Address`
`Group-Base Address Mask`
`Chip-Select`
	Corrupted memory maps making code and data fetches impossible.

`LCD Controller Module`	Affect LCD functionality. It may be possible to cause LCD hardware damage by modifying the refresh frequency or by improper power cycling.

Table 3: Selected registers and theorized effects of improper modification

gether. Placing the onus of allowing or disallowing certain functions on the user can be problematic as, more often than not, the user is not security-conscious and will improperly configure, circumvent, or completely ignore the protection mechanisms due to their complexity. Security processes need to be in place at the operating system level that are undetectable and inescapable.

While this technique of trapping operating system calls has enjoyed some amount of success, it has the drawback that applications legitimately creating and erasing their own databases are often hindered. One remedy to this situation is to have the operating system enforce rules that only allow modification to databases with the same Creator ID as the application performing the actions. In this case, the Creator ID would need to be non-modifiable by the user.

9.2 Register Manipulation

While attacks using the Palm OS API are a major threat, lack of compartmentalization in the operating system allows the user to target the underlying hardware controlling the device. The DragonBall allows direct control of its registers via memory-mapping. Direct control of these registers allows an attacker to control many low-level aspects of device operation. An application simply has to define a pointer to the specific memory location representing the target register.

By examining the DragonBall registers, we have determined particular registers that, when improperly modified, can lead to disruptive events or physical damage to the Palm OS device. Our theorized effects are listed in Table 3. It should be noted that while these examples focus on the DragonBall processor, other embedded microprocessors exhibit similar vulnerabilities. These attacks are comparable to the desktop computer environment in which malicious programs would change the synchronization rate of a monitor or over-drive and manipulate hard drive heads.

9.2.1 Recommendations

Direct register access is not detected by existing anti-virus software. Current software in this field only watches for improper usage of the Palm OS API function calls (such as the `DmEraseDatabase` function).

Discerning a legitimate application from a malicious application is challenging when direct register access is involved. One solution is to prevent any third-party application from direct register access. While this would hinder legacy applications that did not adhere to the published API, the minor loss in backwards compatibility would most likely be deemed acceptable for the increase in security.

9.3 Memory Corruption

Devices using Flash memory supporting field-upgradeable operating systems have inflection points that ROM-based devices do not. Malicious code is capable of taking advantage of the field-upgradeable capabilities of the Flash device to modify or destroy data. Through this, they can patch the operating system with custom code or completely overwrite it. [9, 10] provides details of performing operating system upgrades in the Flash memory of Palm OS devices.

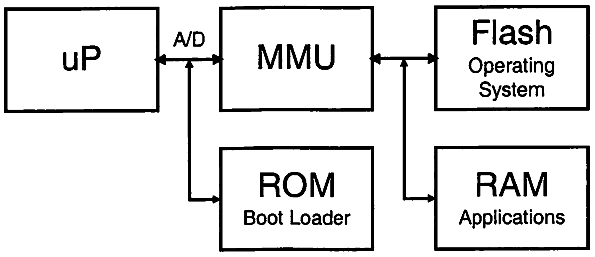


Figure 9: Possible design configuration for a secure PDA

Successful attacks on Flash can be crippling for the Palm OS device. The critical boot loader functionality for controlling field-upgrades is often stored in Flash. If this area is not properly protected using the Software Protection and Boot-Block locking features provided by the Flash memory device, it can be altered. Complete erasure of the boot loader prevents field-reprogramming of the operating system and will require the device to be returned to the factory for replacement. Any data not stored in protected areas of Flash memory is subject to erasure or modification, often without detection.

9.3.1 Recommendations

Current implementations of Palm OS devices do not use any Flash memory for application data storage and is used solely to store the operating system itself. All applications and data reside on battery-backed RAM. Therefore, a trivial solution for security-critical deployments would be to use devices that store the OS in ROM (such as the PalmPilot family) or guarantee that the entire Flash device is read-only. A similar scenario (Figure 9) would be to use a ROM device for all boot loading and Flash memory upgrade routines, still leaving the actual operating system in Flash. This would allow the critical routines to be protected and still allow the OS to be upgraded. It is apparent that the current PDA model places convenience of OS upgrades of greater importance than security.

A disadvantage to using Flash memory for the storage of applications and other often-modified data is the low amount of write-cycles (typically ≈10,000) guaranteed during the memory's lifetime. Given that RAM has no such limitation, it is still a natural choice for this type of data storage.

The Boot-Block areas of Flash memory could be used to implement a secure boot process similar to [1], which will guarantee the integrity of the system.

Implementing a hardware-based memory management unit (MMU) will aid in supplying memory isolation and preventing applications from unauthorized access to external memory. The MMU, commonly designed into embedded microprocessors, is not available in the DragonBall core. For purposes of Palm OS devices, this unit could be implemented in an application-specific IC (ASIC) or programmable logic device. It is hoped that an MMU is designed into the ARM core for future DragonBall processors. The MMU is located on the address and data buses between the microprocessor and the external memory. If the address requested for read/write access is outside of a legal, pre-defined range, the MMU can either prevent the operation outright or respond back to the processor in some manner.

It should be noted that solely implementing an MMU is not enough for proper memory protection. If the Palm OS is modified by an adversary, it may still be possible to access "restricted" areas of Flash. Using [1] in conjunction with an MMU implementation will work nicely, as there is integrity to guarantee that the operating system and underlying components are trusted and there is hardware-based memory protection for fault isolation. Figure 9 is one possible design configuration. The ROM and the MMU could be internal to the CPU, depending on its type. The MMU will monitor the address and data buses as described previously. The entire configuration could be designed as an ASIC or as a secure cryptographic coprocessor, along with the proper tamper-response and physical protection systems as recommended in [4, 6].

Another solution to the problem of accessible Flash memory and risks of intentional corruption would be to introduce hardware jumper protection. This would physically allow or prevent writing to the Flash device. In order to accomplish this, a user would typically have to place a jumper or depress a button to enable or disable writing to Flash memory areas. Such a jumper could be connected to the Chip Enable, Write Enable, or Output Enable line of the memory device. Alternatively, it could enable circuitry that would connect the required address lines between the processor and memory device. When enabling field-upgradeable functionality, some modicum of due diligence must be taken to ensure integrity and authorization for such actions. Even if the hardware jumper was only active for the regions storing the base operating system, this would increase the security of the system. If applications are stored in Flash in future devices, the same scenario would exist and the user would have to physically "approve" each application as it is loaded into their device. This, however, is tedious for the user and could easily be bypassed with simple modifications to the hardware.

Secure coprocessors, such as [8, 26], enable secure distributed applications by providing safe havens where an application program can execute, free of observation and interference by an adversary with direct physical access to the device [26]. Designing such a configuration into the underlying Palm OS hardware will greatly enhance the security of the device and may minimize enough risk to be a suitable platform for security-based applications. It is possible that smartcards can serve as interim cryptographic coprocessors for portable devices [28]. Additionally, [3] proposes a software-based solution of using PDAs as cryptographic tokens.

Currently, Palm OS devices are extremely vulnerable to Flash memory attacks and have no protection mechanisms as described in this section. This is quite possibly the case for other PDAs and portable devices, as well.

10 Conclusions

In this paper, we analyzed the design of the Palm OS and hardware platform with respect to data storage issues, improper security design, and malicious code threats. Vulnerable and at-risk areas were identified that could be taken advantage of for such attacks. It has been pointed out that a variety of problems exist that can be exploited at both the operating system and hardware levels. Specific changes to Palm OS and its associated hardware were recommended and would be required to begin to properly implement preventive measures.

For solutions, it becomes apparent that implementing layer-based access control may be necessary to allow the application level to communicate only with the operating system. Conjunctively, these access control mechanisms would allow the operating system only to communicate with the hardware. The current design of the Palm OS software and hardware is not laid out in this fashion. As a result, many of the attacks discussed in this paper remain extremely difficult to defend against with third-party software running at the application layer. If future versions of Palm OS allow third-party applications to run as multi-threaded, anti-virus applications could essentially run in the "background" and use monitoring techniques as proven useful in desktop environments. Additionally, it may be possible to emulate a virtual machine that provides integrity and memory protection. Virtual memory areas of RAM used during cryptographic operations can be encrypted similar to [24] to protect temporarily stored plaintext.

The cryptographic code signing of applications has been used in many ActiveX scripts and Java applets for a number of years. Portable devices should employ such methods to verify the integrity of trusted applications. Ideally, the code signing routines and resultant signatures would be stored in ROM along with the Certificate Authority (CA) public key of the product vendor. It may be possible to store signatures in Secure Digital (SD) external memory cards (which are planned to be designed into Palm OS devices in late 2001) or Handspring's Springboard modules.

In lieu of any operating system upgrades or hardware re-designs, there are a number of simple and immediate precautionary measures a user can exercise to reduce the risk of data theft or malicious attacks:

- Be aware of what applications are being loaded onto the portable device. If an application comes from an untrusted source, extra care must be taken. This may entail using an existing anti-virus package on the PC to scan the file for known threats or testing the application functionality on a spare device.

- Monitor the HotSync Log and Last HotSync Operation date to verify that there were no unauthorized HotSync operations performed.
- Disable the "Beam Receive" functionality in the System Preferences panel. Enable this feature only when necessary. This prohibits anyone from beaming information to the Palm OS device.
- Be aware of the physical location of your Palm device at all times. Attaching a belt clip or lanyard will reduce loss, misplacement, or theft.

Because Palm OS devices account for the majority of the PDA market, it is hoped that the research in this paper is used to create a more secure computing environment in the short term. It is also hoped that the analyses and ideas provided in this paper will be used in future work to design more secure products.

In the current state, caution should be taken when employing portable devices for security purposes. In a War College-style approach, it is believed by the authors that oftentimes the simple knowledge of a vulnerable area is enough to help steer the user towards more security-conscious use.

Acknowledgments

The authors would like to thank @stake's Research Labs, especially Brian Carrier, for constructive criticism and interesting discussions.

References

[1] W. Arbaugh, D. Farber, and J. Smith, "A Secure and Reliable Bootstrap Architecture," *IEEE Security and Privacy Conference*, May 1997.

[2] ARM, Ltd., "Motorola's DragonBall Processor Portfolio to Include ARM Architecture in 2001," Press Release, December 11, 2000.

[3] D. Balfanz and E. Felten, "Hand-Held Computers Can Be Better Smart Cards," *8th USENIX Security Symposium*, Washington, D.C., August 1999.

[4] D. Chaum, "Design Concepts for Tamper Responding Systems," *Advances in Cryptology: Proceedings of Crypto '83*, 1984.

[5] E. Chien, "Malicious Threats to PDAs & Prototype Solutions," *Virus Bulletin Conference 2000*, September 2000.

[6] A.J. Clark, "Physical Protection of Cryptographic Devices," *Advances in Cryptology: EUROCRYPT '87*, 1988.

[7] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer, "A Secure Environment for Untrusted Helper Applications," *6th USENIX Security Symposium*, San Jose, California, July 1996.

[8] P. Gutmann, "An Open-Source Cryptographic Coprocessor," *9th USENIX Security Symposium*, Denver, Colorado, August 2000.

[9] T. Harbaum, "Flashlib," April 1999, `http://bodotill.suburbia.com.au/flashy/flashy.html`.

[10] T. Harbaum, "OS Flash," September 2000, `http://bodotill.suburbia.com.au/osflash/osflash.html`.

[11] IDC, "Market Mayhem: The Smart Handheld Devices Market Forecast and Analysis, 1999-2004," Report 22430, June, 2000.

[12] I. Jermyn, A. Mayer, F. Monrose, M. Reiter, A. Rubin, "The Design and Analysis of Graphical Passwords," *8th USENIX Security Symposium*, Washington, D.C., August 1999.

[13] E. Keyes, "Hacking the Pilot: Bypassing the Palm OS," *PDA Developers 4.6*, November 1996.

[14] Kingpin, "Palm OS Beam Bit Modification Tool," January 1999, `http://www.atstake.com/research/tools/beamcrack.zip`.

[15] Kingpin, "Palm OS Password Retrieval and Decoding," *@stake Security Advisory*, September 26, 2000, `http://www.atstake.com/research/advisories/2000/a092600-1.txt`.

[16] Kingpin, "Palm OS Password Lockout Bypass," *@stake Security Advisory*, March 1, 2001, `http://www.atstake.com/research/advisories/2001/a030101-1.txt`.

[17] D. Klein, "Foiling the cracker: A survey of, and improvements to, password security," *2nd USENIX Security Workshop*, August 1990.

[18] McAfee.com, "Increased Protection for Wireless Users in Wake of Recent PDA Trojan Discovery," Press Release, September 5, 2001.

[19] United States General Accounting Office, Report to the Chairman, Subcommittee on Telecommunications and Finance, Committee on Energy and Commerce - House of Representatives, "Virus Highlights Need for Improved Internet Management," *GAO/IMTEC-89-57*, June 1989.

[20] Palm, Inc., *Palm OS Programmer's Companion*, DN 3004-003.

[21] Palm, Inc., *Palm OS Programming Development Tools Guide*, DN 3011-002.

[22] Palm, Inc., *Palm OS SDK Reference*, DN 3003-003.

[23] W. T. Polk and L. E. Bassham, "A Guide to the Selection of Anti-Virus Tools and Techniques," National Institute of Standards and Technology Computer Security Division, *SP 800-5*, December 1995.

[24] N. Provos, "Encrypting Virtual Memory," *9th USENIX Security Symposium*, Denver, Colorado, August 2000.

[25] B. Schneier, "The Trojan Horse Race," *Communications of the ACM*, Volume 42, Number 9, September 1999.

[26] S.W. Smith and S.H. Weingart, "Building a High-Performance, Programmable Secure Coprocessor," *Computer Networks (Special Issue on Computer Network Security)*, 31: 831-860, April 1999.

[27] TRG Products, Inc., "FlashPro," `http://www.trgnet.com/cat-flashpro.htm`

[28] University of Michigan, "Smart Card Research At CITI," `http://www.citi.umich.edu/projects/smartcard`.

Secure Data Deletion for Linux File Systems

Steven Bauer, Nissanka B. Priyantha
{bauer,bodhi}@lcs.mit.edu
MIT Laboratory for Computer Science

Abstract

Security conscious users of file systems require that deleted information and its associated meta-data are no longer accessible on the underlying physical disk. Existing file system implementations only reset the file system data structures to reflect the removal of data, leaving both the actual data and its associated meta-data on the physical disk. Even when this information has been overwritten, it may remain visible to advanced probing techniques such as magnetic force microscopy or magnetic force scanning tunneling microscopy. Our project addresses this problem by adding support to the Linux kernel for asynchronous secure deletion of file data and meta-data. We provide an implementation for the *Ext2* file system; other file systems can be accommodated easily. An asynchronous overwriting process sacrifices immediate security but ultimately provides a far more usable and complete secure deletion facility. We justify our design by arguing that user-level secure deletion tools are inadequate in many respects and that synchronous deletion facilities are too time consuming to be acceptable to users. Further, we contend that encrypting file information, either using manual tools or a encrypted file system, is not a sufficient solution to alleviate the need for secure data deletion.

1 Introduction

Secure deletion of data has been considered for years, and different implementations of secure deletion facilities abound. However from our survey of secure deletion techniques, no one yet has implemented it completely and in a truly usable fashion. After we explain our original motivation and background material, we will detail the shortcomings of existing techniques for securely deleting information from disks.

Initially, our motivation stemmed from considering distributed access to sensitive data and distributed file systems. Users increasingly access their data from remote locations including home and office machines, terminals in airports and Internet cafes, and multiple workstations in universities' computer clusters. Many applications and distributed file systems cache sensitive user data on the local disks to improve performance. Distributed file systems such as AFS [11] may cache user data on the client machine or users themselves may copy sensitive files to the local directories. Many web browsers cache accessed information on the local drive. Even when the web cache is cleared the data remains on the underlying physical disk.

Users need assurances that their sensitive data does not remain visible on every machine they use. Sensitive data must be overwritten, sometimes with multiple overwrite passes, making the original data inaccessible even to advanced probing techniques. We have implemented such a secure deletion mechanism in the form of a configurable kernel daemon that asynchronously overwrites disk blocks. The interface to the daemon is general; any block-oriented file system can use the daemon to overwrite blocks on a particular device. Once the overwrite process is complete, the daemon invokes a registered callback that updates the file system state.

Section 2 briefly covers background information explaining how data is stored on disk, how it can be recovered even after it has been overwritten a limited number of times, and requirements for ensuring that deleted data cannot be recovered. Section 3 discusses the shortcomings of user-level secure deletion tools and problems with relying solely upon cryptographic techniques to prevent deleted data from being accessible. Section 4 details the design goals of our system. Section 5 details our implementation and usage suggestions. Section 6 considers the performance, ease of use, applicability, and security of our system. Finally, Section 7 concludes the paper.

2 Background

Recovering data deleted normally from a disk drive is remarkably easy. Most users are not aware that after they delete a file it still remains visible on the disk until overwritten by new data. This may mean deleted data remains on the disk for considerable lengths of time. Many user space tools that recover deleted files exist both for Unix, Windows and Macintosh machines[8]. The ability to recover supposedly deleted files is beneficial to users who inadvertently remove important files, but most people would be shocked to learn that their deleted data is still very accessible.

Even after data has been overwritten it may still be accessible. Magnetic force microscopy and (MFM) magnetic force scanning tunneling microscopy (STM) are two techniques that enable the imaging of magnetization patterns with remarkably high resolutions. Using MFM and a knowledge of well documented disk formats an image can be developed in under ten minutes for older drives[10]. Data is stored on a disk as patterns of varying magnetic strength and each write of the disk head changes the magnetic field strength at a position in a predictable manner. Scanning tools can "peel back" layers of this magnetic information recovering older data. (A much more complete technical description of the process can be found in the references[10, 12].)

On older disks the encoding patterns are referred to as *run-length-limited encodings* (RLL) since they limit the number of consecutive ones and zeros appearing in the encoding patterns. Modern drives use a different encoding scheme called *Partial-Response Maximum-Likelihood* (PRML) encoding. The difference is in the constraints placed on the encoding data patterns. To most effectively overwrite a portion of a disk each magnetic domain on the disk should be flipped a number of times. While older drives could be overwritten more effectively by employing particular overwrite patterns, specific patterns have not been designed for the existing PRML encoding techniques.

There is considerable controversy regarding the capability to recover data that has been overwritten. Prevailing attitudes among some Internet communities seems to be that various government agencies have the ability to recover data from drives even if the data has been overwritten *any* number of times. Numerous anecdotal stories regarding these supposed capabilities can be found on the web. Though still heavily referenced, the Department of Defense standard *DOD 5220.22-M* [16] is outdated and does not reflect current drive technology. It mandates that seven random read/write passes be made over data before it is considered securely destroyed. This compels many users to believe that large numbers of overwrite passes are required.

Twenty commercial data recovery companies were contacted during this project. Each was asked if they could recover a 100KB data file that had been accidentally completely overwritten once with random data. All but one indicated that they could not recover the data if it had really been overwritten. One company [2] that did possess appropriate tools and was willing to try casually estimated the chance of success at "less than one percent."

It is difficult to ascertain what truly is possible. Given the wide variety of opinions and desires, our approach is to let users select the deletion procedures they feel most comfortable using. We strongly suspect that users will have made the recovery task impossible with a small number of overwriting passes at least for modern disk drives. For older drives, additional overwrite passes should be employed[10].

3 Limitations of Alternative Approaches

User-level secure deletion tools and cryptographic techniques attempt to provide some guarantees that sensitive data will not be recoverable once it is deleted. These approaches are useful in many respects, but are not always an appropriate solution. Secure data deletion implemented at the file system level is still required. This section details the limitations of user space tools and cryptographic techniques for secure data deletion.

3.1 User Space Deletion Tools

Any implementation of secure deletion at the user-level will be inadequate. Numerous user-level secure deletion tools already exist (for example [5, 17, 19, 21].) While these tools can be used productively for immediate synchronous overwriting of individual files, they do not provide a complete usable solution. All user space programs face the following problems that cannot be addressed effectively.

1. **File meta-data cannot be overwritten completely at the user level:** The sensitive information in a file system includes the contents of a file and the file meta-data including the name, size, owner information, and creation, modification, access, and deletion times. User-level programs overwrite only the file data itself. Although parts of the meta-data could be overwritten using `touch` to set the file access, creation, and modification times and renaming the file to obfuscate the file name, such techniques are cumbersome and inefficient. Even with such a workaround, important file meta-data information such as what blocks the file contained, user and group ownerships, and deletion time could not be removed.
2. **Secure deletion tools cannot be interposed between all file operations:** Although it is possible to replace obvious file removal programs such as `rm` with a user-level secure deletion tool, this does not work with other less obvious means of deleting file data, such as replacing a file with the contents of another.

   ```
   cp <file> <existingfile>
   ```

   To be used with a user-level secure deletion mechanism, these commands would have to be wrapped with scripts that would first securely overwrite the existing file before proceeding with the normal file operation. Further, and particularly problematic, a user-level deletion tool would have to be integrated with every application that handles its own file management. While dynamically linked libraries could be changed appropriately, statically linked binaries would remain a problem.
3. **File truncation cannot be handled effectively:** User-level deletion programs only overwrite an entire file and thus do not handle file truncation. If someone uses an editor to delete half a file, causing some blocks to be returned to the list of free blocks, that information will not be securely overwritten and will remain visible on the disk. One might imagine handing off a file to a secure deletion tool and telling it to overwrite past a particular offset. However, this interferes with any correctness notions an application might have about file contents and is inefficient if the truncation would not actually return a block to the free list. Fundamentally applications calling `truncate` do not have enough information to know if blocks have to be overwritten. Again, all applications invoking file truncate would have to be modified.
4. **Overwriting data synchronously is inconvenient and often unusable from a user perspective:** When a synchronous deletion tool is used, users are generally unwilling to wait for the overwriting process to complete. While the deletion process could be placed in the background, allowing activities to proceed, application correctness may depend upon the file not being in the name space after the deletion tool is called. The deletion tool could rename a file before the overwrite process begins; however, renaming is not possible for a truncation operation since the file is not removed from the name space.

3.2 Cryptographic Techniques

Another approach to prevent user data from remaining visible on the disk is to use encryption. The possible approaches are to use either an encrypted file system or encryption tools to selectively encrypt files on a disk. The idea is that encoded data will not be accessible without the encryption key. If the encryption key can be intentionally lost, the data is destroyed without overwriting it.

This approach has been proposed [4] for revoking data from both the file system and all backup tapes on which the data is stored. The security of this particular system depends upon either the proper management and destruction of personal copies of a master key or upon a third party trusted to properly destroy their public private key pairs periodically. While being a compelling solution from the standpoint of handling all backup data, the solution is not as satisfying from a practical standpoint. Users who want data to be securely deleted may not want to trust *any* third parties and the alternative of storing and destroying personal copies of encryption keys will seem laborious to many users. We address how backups should be handled in our system in Section 7.

Another cryptographic approach is to use a Steganographic File System[1]. While not attempting to achieve the same goal, deleted data in such a system could not be proved to exist, therefore a user could not be compelled to turn over an encryption key. Secure deletion must mean that even the owner of the data cannot recover it later by any means. There is a psychological need satisfied by knowing that even oneself cannot recover securely deleted data.

In a more general sense, all cryptographic approaches suffer from the following common problems.

1. **Encryption keys can be revoked or compromised:** Anytime that a key is revoked the data associated with it must be considered accessible. Users can be compelled by law to turn over their encryption keys or their keys can be otherwise compromised. In any case, deleted encrypted-files that remain on a disk are as accessible as plain-text if the key is available.

2. **Encryption may not be a viable option for performance or legal reasons:** The performance of an encrypted file system compared to a regular file system may be unacceptable. Encryption tools may be too much of a performance penalty since the tool must be used for every file operation, not just at data deletion time. Users may simply want assurances that their deleted data is not accessible but not want all file data to be encrypted. In some countries, encrypting file data may not be a legal option or available encryption may not be acceptably strong.

3. **Plaintext files may remain visible:** Before files are encrypted, they may have existed as plaintext stored either in temporary or regular files. See the BugTraq archives [3] for an interesting discussion of plaintext temporary files created by the Windows 2000 EFS [14] that remain visible on disk in some common cases. These plain text files must be deleted securely.

4 Design Goals

Our design focused on addressing two main goals: *completeness* and *transparency*. Completeness entails that all the data over the entire course of a file's existence must have been securely overwritten. File data read into application buffers and swapped out to a swap disk must not be accessible either. Security guarantees must be maintained if a system crash happens before the disk blocks have been fully overwritten. If a crash occurs between writing the sensitive data to a disk block and modification of the inode, we need to ensure that the data is properly erased. Thus the procedures followed after crash need to be modified to handle this problem. The goal is for no remnants of a deleted file to remain anywhere on the physical disk.

Our completeness requirement entails that we consider different aspects of a file system usage such as the disk drive behaviors, system administration policies, and user practices. SCSI disks, for instance, keep a large in-memory cache. The proper mode bits must be set on SCSI drive writes to ensure that data actually is written to the physical disk. Drives that remap blocks to other sectors upon sector failure must be addressed. Another issue that must be considered is the possibility of having multiple copies of a file, either because of a users' actions or the backup policies of a system administrator. While our system does not address this facet of completeness, we do our best in our system documentation to ensure that users are aware of all the issues.

Our second goal is to achieve transparency from both a user perspective and system perspective. File deletion and truncation must be very fast to satisfy user expectations. A regular remove operation or file truncation operation can proceed at in-memory speeds. When a user deletes half a document that was previously saved, they do not consider the effects of the underlying disk blocks being returned to a free list. When replacing one file with another, a user does not typically consider the removal of data blocks from the replaced file that actually occurred within the file system. Users thus are accustomed to data deletion operations being fast and transparent.

Secure removal is inherently a slow operation that involves ensuring the underlying disk blocks have been overwritten with data multiple times. Overwriting a file with even two passes of data involves writing the data, flushing the data to disk, waiting for the head seek and writing to complete, and then repeating the process. This will be unacceptably long for even a small file. Were a user to delete a file with megabytes of data, the removal process would be intolerably slow. A user should be able to delete or truncate a file and proceed immediately with other normal operations. To this end, we use an asynchronous deletion process that takes the deleted blocks and writes over them numerous times. As far as a user is concerned, the user time spent in deleting a file securely is comparable to the time taken for deleting a conventional file.

To ensure transparency, disk quotas must be maintained properly during the period of time while the blocks are being overwritten. We require that a user's disk quota reflect the fact that the disk blocks being overwritten are not available for reassignment. Blocks remain a part of a user's disk quota until the overwrite process is complete and the blocks are returned to the free list. Without this requirement, a user would be able to quickly allocate and securely delete large files, starving other users of disk resources.

We provide our system with the caveat that our secure deletion tool intentionally uses deletion techniques that preserves the integrity of the disk drive. For some users, true peace of mind may come only from using a degaussing tool or following other suggested techniques such as burning, or pulverizing the disk[7]. Our secure deletion technique is designed for those interested in leaving their disk in a working state. Anyone requiring more extensive destruction of data and device obviously should pursue other options.

5 Implementation

Our system is split into two parts, a kernel daemon that overwrites blocks on a device, and modifications to a file system that hands off disk blocks to the daemon and appropriately overwrites file meta-data. This is a general design that can support any block-oriented file system. We have implemented the necessary modifications for the Linux Ext2 file system. Other file systems may be added as time permits.

The goal of our modifications is to completely remove the remnants of any file or directory that needs to be deleted securely. When a file is deleted or truncated, we pass the released disk blocks to an asynchronous daemon process that overwrites the data blocks a configurable number of times. Only after the blocks have been overwritten do we return them to the file system to be reallocated. An important benefit of this approach is that the asynchronous daemon can perform the overwrites while the disk would otherwise be idle. Our approach sacrifices immediate security by allowing sensitive data to remain on the disk past the point where the user has deleted it. However it ensures that regular disk operations can proceed without being delayed by the secure deletion of files. In the following sections we explain the modifications to the Ext2 file system code and the implementation of the asynchronous deletion daemon.

We implemented our system using the latest Linux kernel version, which at the time was *linux-2.4.2*. The compiled daemon is $12KB$ in size. The modifications to the file system adds an additional 3900B to Ext2 file system. Overall the code for the overwriting daemon entails roughly 800 lines of kernel code. We have been successfully running this system for the past month on one of the author's machines.

5.1 Secure deletion in Ext2 file system

The Ext2 file system already contains an inode flags field for which one flag is supposed to indicate secure deletion. Secure deletion itself was not previously implemented probably because of the performance penalty of a naive implementation. On Linux, various file flags can be listed using the `lsattr` command and set using `chattr`. The secure deletion flag is set by issuing the

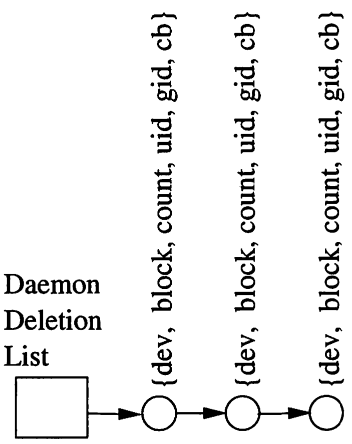
```
chattr +s <filename>
```

command. Directories as well as files can be marked for secure deletion. Any file created in a directory marked for secure deletion will have the secure deletion flag set. This inheritance of flags is beneficial since a user can mark entire trees in the file system name space for secure deletion where all files created on the branches will have the secure deletion flag set. Flags are preserved for most typical file operations. Users should be aware that the copy command does not copy the file flags. (The new file would have the secure deletion flag set if it was created in a directory marked for secure deletion.)

5.2 Ext2 Modifications

We had to make few changes to the Ext2 code to implement secure deletion. We made seven modifications to the existing code base. These can be seen in Table 1. In each function we tested whether the secure deletion flag was set. If it was not set then function behavior proceeded normally. If the secure deletion flag was set then we modified the code to implement the overwriting process. When data blocks that need to be overwritten are released from a file, we add an element containing a tuple of device identifier, beginning block number, number of blocks released including this block number, the user and group identifiers, and the function to be called once the blocks have been overwritten to the daemon deletion list (Figure 1).

We pass the user and group identifiers to the deletion daemon so that the correct disk quota can be maintained at all times. After the disk block has been returned to the free list, the disk quota system is updated using these identifiers. We use the *(uid, gid)* pair to maintain the number of blocks belonging to the pair currently being overwritten. Once the blocks have been overwritten, we then call the disk quota system with a VFS inode dynamically created with the proper user, group, device, and block count information. This does not interfere with the disk



Daemon Deletion List
{dev, block, count, uid, gid, cb}
{dev, block, count, uid, gid, cb}
{dev, block, count, uid, gid, cb}

Figure 1: The deletion list data structure holding the data blocks removed from files

quota inode count since we are not freeing inodes but only disk blocks.

In our current implementation the file metadata for securely deleted files and directories is overwritten once synchronously with all zeros within the Ext2 code itself. We made this decision since it was simpler to implement at the time. Also, currently between the time when blocks are initially passed to the deletion daemon and the time they are returned to the free list, the blocks are not associated with any file in the file system. This complicates the recovery process if a crash occurs during this time. Future modifications will add the blocks to files in a directory storing all the blocks that must be overwritten.

5.3 Ksdeletion Daemon

The Ksdeletion daemon starts at the system initialization time or when the secure deletion module is loaded. It is a kernel daemon that runs at periodic intervals. Behavior of the daemon is dynamically configurable through a `/proc` file system interface. Every time the daemon wakes up, it performs three tasks: (1) retrieves and stores the blocks that need to be overwritten (2) issues new overwrite requests and (3) returns blocks that have been fully overwritten to the corresponding file system for reallocation. A file system is prevented from being unmounted if its blocks are in the process of

ext2_free_blocks	modified to place blocks on daemon secure deletion list, call disk quota system freeing these blocks from the current inode, call disk quota system adding blocks back to the unique user and group inode
ext2_free_clean_blocks	renamed regular ext2_free_blocks
ext2_discard_prealloc	modified to call ext2_free_clean_blocks
ext2_delete_inode	modified to reset all inode values to zero
ext2_unlink	modified to overwrite the name of a file from the directory
ext2_rmdir	modified to overwrite the name of the directory in the parent directory
ext2_free_overwritten_blocks	new function, returns overwritten blocks to the free list and updates disk quota

Table 1: Modifications done to Ext2 file system for implementing the secure deletion

being overwritten.

Each device contains a generations data structure. Each generation contains two lists, one list for disk block information in the form of *(block, count)* groups and the second list containing user information in the form of *(uid, gid)* pairs. Different generations represent lists of blocks that have proceeded through the overwriting process a different number of times. The number of generations used depends upon the overwrite strategy employed. If the policy is to completely overwrite a set of blocks multiple times before proceeding to the next set, then only two generations are required. One generation will represent the blocks currently being overwritten while the other generation will represent the blocks that have never been overwritten. Another policy might mandate that blocks be overwritten as soon as possible. To accommodate this type of policy we allow for a configurable number of overwrite generations. As blocks are progressively overwritten they are moved further in the generation data structure. New blocks are always added in the first generation slot. Figure 2 presents a diagram of this data structure.

The daemon retrieves newly deleted blocks that the file system has added to the daemon deletion list and places them in the above data structure. The storage of the blocks on the daemon deletion list is inefficient since each *(block, count)* group requires its own list element. No aggregation of blocks across the entire device is possible in this format. But when the daemon adds the elements from the deletion list to the generation data structure, disk block information of adjoining disk blocks (belonging to the same generation) are merged to enable an efficient representation. If a new set of blocks cannot be merged, a new element is created at the proper place in the list to ensure the correct ordering of disk block numbers. Since the list is ordered by the actual block number, it is easy to achieve very efficient overwriting of data especially when a large number of disk blocks are to be overwritten. We also add a *(uid, gid, count)* element to a linked list of user information if the *(uid, gid)* pair is new, or increment the count of an existing *(uid, gid)* pair.

The second task of the daemon is to issue new device write requests. Each time the daemon is activated it runs through all the devices writing out a configurable number of disk blocks. Before issuing write requests the daemon checks the number of outstanding device requests. If the number of outstanding requests (read or write) exceeds a configurable level then no new requests are issued for that device. One side note is that the kernel interface to the block devices extensively makes use of *buffer_head* data structures, but operations on the buffer head memory cache are not exposed. It would be beneficial if they were since then we could reuse the free buffer head pool already available. Since this is not exposed, we maintain our own buffer head cache that employs the underlying slab allocation memory routines. This is done to improve the efficiency of memory allocation.

Finally, the third task of the daemon is to return blocks that have been securely overwrit-

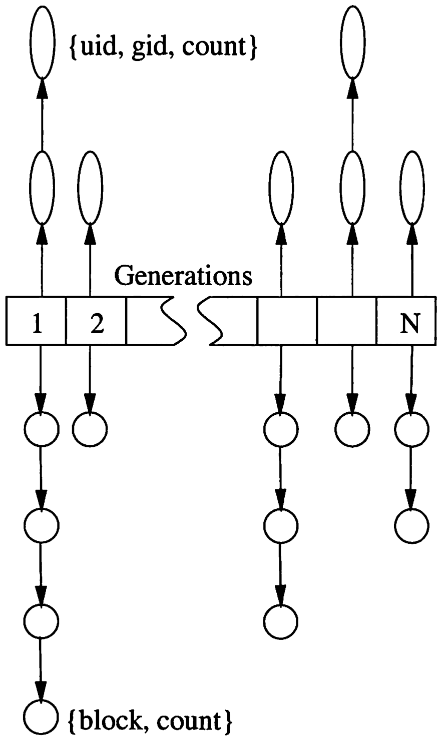


Figure 2: The generation data structure kept by secure deletion daemon

ten to the file system by invoking the registered callback of the file system. All the blocks in one generation are returned to the free list at the same time. The disk quota system then is properly updated using the *(uid, gid, count)* list.

5.4 Overwrite Policies

Currently the daemon overwrites a configurable number of blocks on each device for each iteration. Another policy being implemented is to overwrite either a minimum number or a percentage of outstanding blocks, whichever is larger. Other policies that could be implemented include overwriting blocks at a rate determined by the amount of free space remaining or policies that make more sophisticated attempts at avoiding regular disk activity.

Parameters of an overwrite policy are dynamically configurable through the /proc file system. For the default policy, parameters include maximum number of blocks written at a time, overwrite patterns employed, number of overwrite passes, and the interval between passes. These parameters are set by writing commands of the following form to `/proc/secdel/policy`:

```
secdel <device> <parameter> <value>
```

5.5 Failure Recovery

A file system may crash before data blocks are overwritten and returned to the free list or sensitive data may be written to a block but not yet be pointed to by a file system element. In either case the inconsistencies that appear in the file system are blocks and inodes that are marked as allocated but are not linked to the rest of the file system. After such a crash the system administrator determines if allocated blocks should be securely overwritten or if they belong to existing files. While still needing to handle blocks that are written but not yet pointed to by a file system, our planned modifications, adding blocks being securely overwritten to temporary files, will make the recovery task easier.

5.6 Backup Copies

We imagine that many users will not need to worry about additional copies of their sensitive data. In many cases backup mechanisms are not employed. In other instances, no backup procedure is necessary since user data is only temporarily accessed and stored on the device (for instance on terminals in Internet cafes or academic computing clusters.) If a user does make backup copies, we suggest that separate devices be used to store data which may need to be securely deleted. Other handling procedures should be instituted for these devices; for instance, backup tapes may be destroyed after a shorter time period.

5.7 Swap Space

A number of approaches exist so that sensitive data is not stored in the swap space on a disk. The easiest solution is to ensure that sensitive data never gets written to the swap disk in the first place. Given the availability of large amounts of memory, the swap-file could simply be disabled. Slightly more complex, an encrypted swap space using a random key could be used. If a longer window of vulnerability is tolerated, the swap-file could be disabled, overwritten and then re-enabled periodically. devices

6 Evaluation

We evaluate our security mechanism using a number of criteria including performance, ease of use, applicability and security. Each of these criteria is important considering the target population for our tool.

6.1 Performance

The impact on user visible latency of our kernel daemon is negligible. Awakened at periodic intervals, the daemon returns immediately if no blocks need to be overwritten. Similarly, the performance impact of our modifications to the Ext2 file system is not detectable by our tests. Measurements of the time taken to return from a user space deletion and truncation were not affected by the addition of our code.

According to [6] file deletion and truncation are less than three percent of total file system operations on a variety of system platforms and workloads. In most cases, these activities constitute far less then one percent of disk operations. Their study does not characterize the amount of data deleted with each operation which is pertinent to our overwrite process. This still tends to indicate that our daemon will be inactive for a majority of the time.

We did not explore whether disk fragmentation is increased as a result of holding blocks for the overwriting process or whether background disk writes have any effect on other user disk activity times. Conceivably, seek times could be increased since the disk read write head will be moved more frequently to other sections of the disk. Another pertinent performance question is the effect of our overwriting process on disk lifetimes since we are increasing the amount of data written to disk. We suspect that this would not be much of a problem given the infrequency of file deletion and truncation suggested by[6].

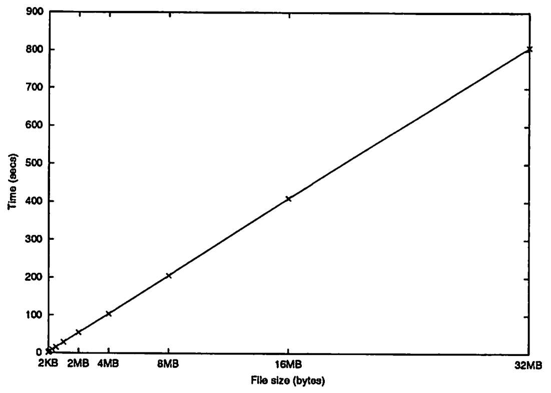


Figure 3: Secure file deletion times for a user-level deletion tool

6.2 Ease of Use

Our secure deletion tool is relatively easy to install since both the daemon and the modified file systems can be loaded as kernel modules. A new kernel does not need to be recompiled. Overwrite policies and their associated parameters can easily be configured through the `/proc` file system interface. Users then only need to make a one-time decision regarding which directories or files should have their blocks securely deleted. In Ext2, entire directory trees can be flagged for secure deletion by issuing one command. From a user perspective, all file operations and application behaviors proceed normally.

To compare the ease of use of our tool with user space tools, we examined how long a typical user space program took to overwrite files of various sizes (Figure 3). For this test, we selected `overwrite` [17], which appears to be a popular secure deletion tool. It uses Peter Gutmann's overwrite patterns [10] to overwrite files thirty-two times. The bandwidth of our disk drive as determined by the hdparm benchmark was $2.52MB/sec$. (It is a relatively old IDE drive.) We believe that even a small file of $256KB$, that takes roughly 8 seconds to overwrite, is too much of a performance penalty to be acceptable if employed frequently. Users definitely would not tolerate waiting 13 minutes for larger file sizes to be deleted.

6.3 Applicability

Any block-oriented file system can use the asynchronous overwrite functionality provided by our deletion daemon to securely delete blocks of data. Log structured file systems, file systems which write redundant data, such as RAID-based file systems, and drives which hide failures through dynamically remapping blocks are not supported. Systems employing backup strategies were addressed in a previous section. We provide warnings with our system documentation so that users are aware when this mechanism should not be utilized.

6.4 Security

The security of our approach depends on the effectiveness of the overwriting techniques employed and the window of opportunity between when the request arrived for the secure deletion and when the overwriting process actually completes. While we strongly suspect that even a small number of overwrite passes will make recovery of deleted data impossible for most modern drives, we cannot know for sure. Users who feel they need more security can configure their installation to employ more overwrite passes.

We can approximate the time taken to completely overwrite a file in our implementation using the following formula assuming that writes from one iteration are complete before the next iteration begins.

Let:
T - *time taken to securely overwrite the file*
P - *probability of being too busy when the daemon wakes up*
F - *file size in bytes*
B - *device block size in bytes*
N - *maximum number of blocks written per iteration*
G - *number of different overwriting passes*
I - *time interval between issuing overwrites*
Then;

$$T = G \times I \times \frac{1}{1-P} \left\lceil \frac{\lceil \frac{F}{B} \rceil}{N} \right\rceil$$

Fig 3 presents sample overwrite times assuming a device block size of 4096B, 1024 blocks written per iteration, a 10 second interval between overwrites, and a zero probability of the

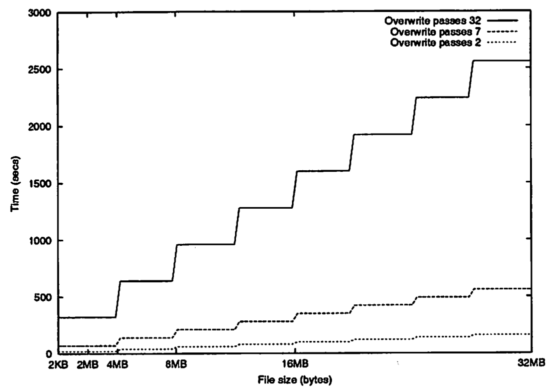


Figure 4: Time to securely delete files for number of overwrite passes

request queue being too busy when the daemon checks. Assuming a zero probability of being too busy makes the results presented a best-case scenario. The overwrite policy employed is to completely finish one overwrite pass before proceeding to the next. This results in the step behavior demonstrated in the graph.

7 Conclusion

This work presents a secure data deletion mechanism for the Linux Ext2 file system. We asynchronously overwrite data according to best known practices. We securely overwrite both the file data and meta-data. Our system is configurable to user needs and the overwriting can take place during periods when the disk would otherwise be inactive. We have argued why existing approaches such as user-level deletion programs and encryption based approaches are inadequate. Perhaps the most compelling argument for our approach is that it is very simple from a user's standpoint. All users have to do to securely delete their data is to set a flag bit on the file or the directory containing that file.

Other things to consider as we develop our system more fully are the primitives that the device drivers offer that we can employ to our advantage. For instance, being able to control the frequency and intensity of disk head signal as the patterns are written may improve the overwrite effectiveness[20]. We also might considerably improve the overwrite efficiency by using rotational latency periods for our passes[13]. Another possible implementation revision is to employ the techniques described in [9] for updating file system meta-data.

We are aware that this tool potentially benefits people who we might not be interested in helping. In discussing this project with colleagues, concerns were raised that it would help hackers hide their activities and allow criminals to evade the law more easily by providing readily available tools for removing evidence both on their own and compromised systems. In many ways these concerns are similar to objections about the wide spread use of encryption. The technology can be used for both good and bad purposes. We strongly suspect that it will give peace of mind to far more people then the number who will use it to evade the law. Similarly, we recommend to those concerned with the human impact of such a tool the article *"In Defense of the DELETE Key"*[18]. It argues that the disk drive is an electronic recording device present in every office and home that records all our written thoughts. If users change their mind later and delete what they wrote, the file actually should be deleted. The author provides convincing examples of why everyday users would want 'delete' to really mean delete.

One final caveat to the user would be that we do not know how securely we have removed data from the physical disk. We use the best known practices available today. Years into the future, this information potentially could be recovered using more advanced techniques. As recording mediums change, the methods for securely deleting data may need to be modified. We were reminded of this fact recently upon learning that the National Archives was investigating whether the deleted sections of the famous Watergate tapes could now be recovered[15].

8 Availability

Our code, installation and usage instructions is available under GPL at the following location:

`http//atlas.lcs.mit.edu/securedeletion`

References

[1] Anderson, Needham, and Shamir. The Steganographic File System. In *IWIH: International Workshop on Information Hiding* (1998).

[2] Authentec International. Private communications.

[3] Berglind, R. BugTraq: EFS Win 2000 flaw. Post to bugtraq mailing list. http://archives.linuxbe.org/arch051/0037.html, January 2001.

[4] Boneh, D., and Lipton, R. A Revocable Backup System. In *Proceedings of the Sixth USENIX Security Symposium* (1996).

[5] CyberScrub Secure File Deletion / Internet Privacy Utility. http://www.cyberscrub.com.

[6] D. Roselli, J. Lorch, T. A. A Comparison of File System Workloads. In *Proceedings of USENIX Annual Technical Conference* (2000).

[7] Magnetic Tape Degausser. NSA/CSS Specification L14-4-A, 31 October 1985.

[8] Ext2fs Home Page. http://e2fsprogs.sourceforge.net/ext2.html.

[9] Ganger, G. R., McKusick, M. K., Soules, C. A. N., and Patt, Y. N. Soft updates: a solution to the metadata update problem in file systems. *ACM Transactions on Computer Systems 18*, 2 (2000), 127–153.

[10] Gutmann, P. Secure Deletion of Data from Magnetic and Solid-State Memory. In Proceedings of the Sixth USENIX Security Symposium, pages 77-89, 1996.

[11] Howard, J. H. An overview of the andrew file system. In *Proceedings of the USENIX Winter 1988 Technical Conference* (Berkeley, CA, 1988), USENIX Association, pp. 23–26.

[12] J-G Zhu, Y. Luo, J. D. Magnetic Force Microscopy Study of Edge Overwrite Characteristics in Thin Film Media. *IEEE Trans.on Magnetics 30*, 2 (1994), 4242.

[13] Lumb, C., Schindler, J., Ganger, G., Nagle, D., and Riedel, E. Towards higher disk head utilization: Extracting free bandwidth from busy disk drives. In *Symposium on Operating Systems Design and Implementation* (October 2000).

[14] MICROSOFT. Encrypting File System for Windows 2000. http://www.microsoft.com/windows2000.

[15] NARA. NARA News Release 00-105. http://www.nara.gov/nara/pressrelease/nr00-105.html.

[16] National Industrial Security Program Operating Manual. DoD 5220.22-M, January 1995.

[17] Overwrite, Secure Deletion Software. http://www.kyuzz.org/antirez/overwrite.

[18] In Defense of the DELETE Key. *The Green Bag 3*, 4 (2000).

[19] Shred Info Entry. Linux INFO tree, File: fileutils.info Node: shred invocation.

[20] T. LIN, J. CHRISTNER, T. M. Effects of Current and Frequency on Write, Read, and Erase Widths for Thin-Film Inductive and Magnetoresistive Heads. *IEEE Trans.on Magnetics 25*, 1 (1989), 710.

[21] Wipe: Secure File Deletion. http://wipe.sourceforge.net/.

RaceGuard: Kernel Protection From Temporary File Race Vulnerabilities

Crispin Cowan, Steve Beattie, Chris Wright, and Greg Kroah-Hartman

WireX Communications, Inc. http://wirex.com/

Abstract

Temporary file race vulnerabilities occur when privileged programs attempt to create temporary files in an unsafe manner. "Unsafe" means "non-atomic with respect to an attacker's activities." There is no portable standard for safely (atomically) creating temporary files, and many operating systems have no safe temporary file creation at all. As a result, many programs continue to use unsafe means to create temporary files, resulting in widespread vulnerabilities. This paper presents RaceGuard: a kernel enhancement that detects attempts to exploit temporary file race vulnerabilities, and does so with sufficient speed and precision that the attack can be halted before it takes effect. RaceGuard has been implemented, tested, and measured. We show that RaceGuard is effective at stopping temporary file race attacks, preserves compatibility (no legitimate software is broken), and preserves performance (overhead is minimal).

1 Introduction

Attacks exploiting concurrency problems ("race vulnerabilities") are nearly as old as the study of computer system security [1, 5]. These are called TOCTTOU ("Time of Check To Time Of Use") errors [6]. Of particular interest is the temporary file creation vulnerability: programs seeking to create a temporary file first check to see if a candidate file name exists, and then proceed to create that file. The problem occurs if the attacker can *race* in between the file existence check and the file creation, and the attacker creates the file that the victim program expected to create.

In concrete terms, this problem occurs on UNIX systems when programs use `stat()` or `lstat()` to probe for the existence of files, and `open(O_CREAT)` to create the files. An encapsulated means to create temporary names is the `mktemp()` library function.[1] The `mktemp()` library function simply encapsulates the `lstat()` call, and thus `mktemp()` followed by `open(O_CREAT)` is vulnerable to race attacks.

This *race condition* becomes a security vulnerability if the victim program creating the temporary file is privileged (i.e. running as `root` or some other privileged user-ID) and the attacker creates a link pointing to a security sensitive file such as `/etc/passwd` or `/etc/hosts.allow`. When this occurs, the `open(O_CREAT)` will obliterate the data contained in the sensitive file. The `fopen()` library function, being a wrapper around `open(O_CREAT)`, is similarly vulnerable.

There are two commonly accepted mechanisms that exist to prevent this race condition: using `open()` with the `O_CREAT` and `O_EXCL` flags, or using the `mkstemp()` library function (which is a wrapper around `open(O_CREAT|O_EXCL)`). When `open(O_CREAT|O_EXCL)` is called on a file that already exists, it will fail and prevent the race attack. Unfortunately, because these mechanisms are not ubiquitously available and portable, common programs (such as Apache [3, 13]) still continue to use `mktemp()` and friends, despite the fact that the Linux `mktemp` man page says "Never use mktemp()."

This paper presents RaceGuard: a kernel enhancement that detects attempts to exploit temporary file race vulnerabilities, and does so with sufficient speed and precision that the attack can be halted before it takes effect. RaceGuard functions by detecting the change in circumstances between the stat() call and the open() call. If the stat() "fails" (the file does not exist), then RaceGuard caches the file name. If a subsequent open() call provides the same name, and discovers that the file *does* exist, then RaceGuard detects a race attack, and aborts the open() operation.

The rest of this paper is organized as follows.Section 2 elaborates on the temporary file vulnerability issue. Section 3 presents the RaceGuard design and implementation. Section 4 presents our security testing against known race vulnerabilities in actively used software. Section 5 presents our compatibility testing, showing that RaceGuard protection does not interfere with normal system operations. Section 6 presents our performance testing, showing that the performance costs of

1. and related library functions tmpnam() and tempnam().

This work supported in part by DARPA contract N66001-00-C-8032.

RaceGuard protection are minimal. Section 7 describes related work in defending against temporary file race vulnerabilities. Section 8 presents our conclusions.

2 Temporary File Race Vulnerabilities

The basic form of a temporary file race vulnerability is that a privileged program first probes the state of the file system, and then based on the results of that probe, takes some action. The attacker can exploit the vulnerability by "racing" between the probe and the action to change the state of the file system in some critical way, such that the victim program's action will have an unintended effect.

The simple form of this attack is temporary file creation. The victim program seeks to create a temporary file, probes for the existence of the file, and if the nominated file name is not found, proceeds to create the file. The attacker exploits this by creating either a symbolic link that matches the name of the file about to be created, and points to a security sensitive file. The result is that the victim program will unwittingly over-write the security sensitive file with unintended content.

A variation on this scheme is the "dangling symlink". The victim program performs the same sequence as above. The attacking program races in and creates a symlink or hard link from the matching name to a *non-existent* file whose existence has security implications, such as `/etc/hosts.allow` or `/etc/nologin`.

Another variation is the "file swap." Here the victim program is a SUID root program that can be asked to write to a specific file [6]. The victim program defensively checks to see if the requesting user has access to the file, and then only does the write if the user has permission. The attacker provides a file that they have access, to, and between the access check and the write operation, the attacker swaps the file for a symlink pointing to a security sensitive file.

3 RaceGuard: Dynamic Protection from Race Attacks

RaceGuard detects attempts to exploit race vulnerabilities at run time by detecting a change in the environment between the time the program probes for the existence of a file, and the time it tries to create it: if the file named "foo" does *not* exist at the time of the stat, but *does* exist at the time of the open, then someone tried to race us, so abort the operation. RaceGuard achieves this by caching the file names that are probed, and when creation attempts occur that hit existing files, the names are compared to the cache. Section 3.1 describes the RaceGuard algorithm. Section 3.2 describes the RaceGuard implementation and the cache management policy.

3.1 RaceGuard Design

RaceGuard seeks to detect pertinent changes in the file system between the time an application probes for a nominated temporary file name, and the time the file is actually created. "Pertinent" means changes with respect to the nominated name. The RaceGuard algorithm to achieve this is as follows:

- Each process keeps a cache of potential temporary file races. This cache is a list of file names, associated with each process control block within the kernel.
- If file probe result is "non-existent file," then cache the file name in the process's RaceGuard cache.
- If file creation hits a file that already exists, *and* the name matches a name in the RaceGuard cache, then this is a race attack: abort the open attempt.
- If file creation succeeds without conflicts, and matches a name in the RaceGuard cache, then clear that entry from the cache. This prevents "false positive" RaceGuard events when a program uses the same name for a file more than once.

This caching mechanism serves to detect and differentiate between the sequence "probe; create", and "probe; attacker meddling; create". To defend against the "dangling symlink" variant attack described in Section 2, RaceGuard does *two* resolves on the name provided to open that are in the RaceGuard cache: the first follows symlinks, while the second does not. If the two resolve differently, and the argument name matches an entry in the RaceGuard cache, then this is treated as a race attack.

RaceGuard does not defend against the "file swap" attack. Because the attack concerns an already existent file, this is not really a temporary file race attack. In practice, such vulnerabilities appear to be relatively rare: searching Securityfocus.com's vulnerability database [18] for "race" produced 75 hits, while searching for "race & !tmp & !temp" produced only 24 hits. Even among the 24, random sampling indicates that many of them are actually temporary file issues, but did not say so in the name of the vulnerability.

3.2 RaceGuard Implementation & Cache Management Policy

The RaceGuard implementation is in the kernel, facilitating both per-process and inter-process RaceGuard cache management. RaceGuard mediates three basic types of system calls:

- those which can inform the program that a file system entry does not exist -- `stat()`, `lstat()`, `access()`, `newstat()`, and `newlstat()`.
- those which enable the program to actually create file system entries -- `open()`, `creat()`, `mkdir()`, `mknod()`, `link()`, `symlink()`, `rename()`, and `bind()`.
- those which create and remove processes -- `fork()` and `exit()`.

These system calls are often called indirectly via library wrappers. For example, an insecure program may use the C library function `mktemp()`, a wrapper for `lstat()`, followed by `fopen()`, a wrapper for `open()`. Placing RaceGuard mediation in the kernel provides protection for such a programs, in an effort to provide mediation of temporary file creation that is as complete as possible [17].

The interesting part of RaceGuard's implementation is the cache management policies: when to place a cache entry, when to clear it, and the cache replacement policy. We take an aggressive position on cache clearing, and a conservative position on cache populating. This results in some potential race vulnerabilities getting past RaceGuard, in exchange for assuring that no legitimate software is disrupted by RaceGuard. We do this because RaceGuard is an intrusion *rejector* in addition to an intrusion detector, making false positives much more critical than false negatives.

The RaceGuard cache is small (7 entries per process) to keep the kernel memory footprint small, as there is one cache per process, one cache entry per file, and each cache entry is large (`MAX_PATH_LEN`). We hypothesize that most race situations occur with little file system activity occurring in the process between the stat() and the open(), thus a small cache will be sufficient.

The assumption that programs will do the probe and creation in close sequence also affects our cache eviction policy. We considered using LRU (Least Recently Used) and FIFO (First In, First Out). LRU is not appropriate because the expected use is one creation and one reference, so a recent reference is not a good basis for retention. We settled on this fast approximation to FIFO:

1. The cache is a circular buffer.
2. Scan the cache for empty slots, and take the first empty slot found. *Note*: empty slots occur naturally for RaceGuard because of the heavy use of cache invalidation upon successful creation of temporary files (see Section 3.1).
3. The above scan is started from the most recently created entry. If no empty slots are found, then eject the entry just before the most recently created entry slot in the circular buffer.

This cache eviction policy is fast, and avoids the pathology of evicting the most recently created entry.

Races sometimes occur between processes, especially for shell scripts. RaceGuard partially deals with this by inheriting the cache from parent to child (which is why `fork()` is mediated). If the parent tested a file's existence with a common shell built-in function such as `[ -f tempfile ]`, this information is shared with its subsequent child processes. Employing our aggressive cache clearing policy, child processes clearing entries from their cache notify their parent to also clear entries. Likewise, children do not try to populate their parents' cache as this would violate our conservative cache population policy and could pollute the parent's cache or cause false positives.

Some system calls which create file system entries are not subject to race conditions because they fail when the entry already exists, i.e. `mkdir()`, `link()`, etc. However, we clear matching cache entries on *any* successful file system entry creation, even those which we do not need to monitor for races. Similarly, many system calls return `ENOENT` informing the user that no file system entry exists. However, we have carefully selected a small subset of these calls to mediate based on real world code. It is common for applications to use `stat()` or `access()` to check for a file's existence, while it is uncommon for applications to use `chmod()` for such a check. This conservative approach to cache population also helps ensure the cache is not polluted.

This approach of cautiously only caching entries from a few file probing system calls is largely effective. However one pathological case exists: when a shell script executes a program, the shell typically `stat`'s for that program file in every directory in the `$PATH`. If the script probes many directories before finding the file it is looking for, it has the effect of flooding the RaceGuard cache with useless entries. Thus, a shell script that probes for a file, executes an external program, and then creates the file, may not be protected by RaceGuard. In our experience, most shell scripts find the executables they are looking for in `/bin` or `/usr/bin` so this problem does not occur in practice.

Note that this problem does *not* occur for native programs and dynamic linking. The GNU/Linux `ld.so` loader uses `open()` rather than `stat()` when searching the `$LD_LIBRARY_PATH` for a `.so` file. The

```
 <<< set up our target to overwrite >>>
[steve@reddwarf .elm]$ echo "please dont hurt me" > ~/dont_hurt_me

        <<< run our vulnerable program >>>
[steve@reddwarf .elm]$ LD_PRELOAD=~/libmktemp.so rcsdiff -u elmrc > /dev/null
===================================================================
RCS file: RCS/elmrc,v
retrieving revision 1.3
unsafe_mktemp[20038]: ImmunixOS unsafe mktemp - about to pass back /tmp/T0LZ388D
        <<< In another shell, do "ln -s ~/dont_hurt_me /tmp/T0LZ388D" >>>
diff -u -r1.3 elmrc

        <<< and what does our target now contain? >>>
[steve@reddwarf .elm]$ cat ~/dont_hurt_me | head -5
#
# .elm/elmrc - options file for the ELM mail system
#
# Saved automatically by ELM 2.5 PL1 for Steve Beattie
#
```

Figure 1 Successful Attack Against RCS Without RaceGuard

execlp/execvp system calls search the path, but they do not stat files; instead, they call execve (the system call), and if it fails they move on to the next directory in the path.

4 Security Testing

Rigorous testing of temporary file race vulnerabilities is problematic, because the vulnerability is fundamentally non-deterministic: the outcome of the attack depends on whether the victim program or the attacking program wins the race to the file in question. Therefore, the security testing in this paper will not be as cleanly definitive as we would like. To do deterministic, repeatable testing, we had to create a situation in which the attacker would *reliably* win the race. We did this by creating a doctored version of the mktemp library call that does two key things:

Pause the Program: our doctored mktemp function pauses the caller for 30 seconds, giving the attacker ample time to deploy the race attack.

Print the Created File Name: The file names produced by mktemp are easy enough to guess that a determined attacker can get a hit and violate security, *eventually*. Our doctored mktemp function shortens this task by printing the name of the temporary file that it will create to syslog. This allows the attacker to precisely deploy a race attack, rather than repeatedly guessing the file name.

While we recognize the limited value of security testing against such a straw-man, we felt it necessary to get repeatable experiments. We view the above concessions as largely immaterial to the validity of RaceGuard defense, because they only make the programs *more* vulnerable. However, it is interesting to note that while exploits for buffer overflow [9], format bug [7], and CGI [8] vulnerabilities are readily available, exploits for race vulnerabilities are extremely rare. We conjecture that the relative scarcity of race exploits is related to the relative difficulty in successfully deploying race attacks: "script kiddies" aren't interested in attacks that are hard to do, and so race attacks remain the purview of the relatively serious attacker.

Using this doctored mktemp function, we attacked four programs: RCS version 5.7 [15], rdist Version 6.1.5 [14], sdiff - GNU diffutils version 2.7 [2], and shadow-utils-19990827 [12]. In each case, without RaceGuard protection, we succeeded in duping the victim program into over-writing an unintended file. Figure 1 shows such a successful attack against RCS. With RaceGuard protection, the identical attack produces a RaceGuard intrusion alert and aborts the victim program, while the file that would have been over-written is unharmed, as shown in Figure 2.

5 Compatibility Testing

RaceGuard is intended to be a highly transparent security solution, which means that it may not break much (if any) legitimate software that is not being actively

```
 <<< set up our target to overwrite >>>
[steve@kryten .elm]$ echo "please dont hurt me" > ~/dont_hurt_me

       <<< run our vulnerable program >>>
[steve@kryten .elm]$  LD_PRELOAD=~/libmktemp.so rcsdiff -u elmrc > /dev/null
===================================================================
RCS file: RCS/elmrc,v
retrieving revision 1.3
unsafe_mktemp[1456]: ImmunixOS unsafe mktemp - about to pass back /tmp/T0POjIdZ
       <<< In another shell, do "ln -s ~/dont_hurt_me /tmp/T0POjIdZ" >>>
/usr/bin/co: Killed
rcsdiff aborted

       <<< and what does our target now contain? >>>
[steve@kryten .elm]$ cat ~/dont_hurt_me
please dont hurt me

       <<< RaceGuard intrusion alert in syslog >>>
[steve@kryten .elm]$ dmesg | tail -1
Immunix: RaceGuard: rcsdiff (pid 1458) killing before opening /tmp/T0POjIdZ!
```

Figure 2 Failed Attack Against RCS With RaceGuard

subjected to actual race attacks. To test this compatibility requirement, we exercised RaceGuard under a wide variety of software. To that end, RaceGuard has been running on various developers workstations day-to-day since January 1, 2001. This section describes the various compatibility faults induced by the original RaceGuard design, and how we addressed them. The current implementation exhibits no known compatibility faults.

The first problem we encountered was manifested by the Mozilla web/mail client. Mozilla makes heavy use of temporary files for caching web content. Re-use of some of these names induced false positive reports from RaceGuard. This problem is what spurred us to add the cache clearing feature, where RaceGuard cache entries are flushed when the corresponding file creation succeeds.

A related problem was induced by the script Red Hat Linux uses to preserve `/dev/random`'s entropy pool across re-boots. This is a shell script in which the parent process does the probe, and a child process creates the temporary file. Adding the feature where clearing the cache entry from a process also clears the entry from its parent's cache (see Section 3.2) fixed this problem.

The third problem encountered was induced by CVS [4] checkout. Here, CVS frequently probes for the same file name in various directories. The rough sequence of "`probe("foo"); chdir("bar"); creat("foo")`" induced a false positive RaceGuard event for the file "`foo`". Changing RaceGuard cache entries from simply the name presented to each system call to a fully resolved absolute path addressed this problem.

Finally, the VMWare virtual machine emulation system [11] manifested a minor compatibility problem with RaceGuard. Portions of the VMWare system periodically make calls to the `stat()` system call with a *null* argument for the pathname, i.e calling "`stat("")`" which is meaningless. Initially, RaceGuard reported a debugging error when this occurred (thinking that it was some kind of error copying syscall arguments to kernel space. However, once we satisfied ourselves that this behavior is harmless, we disabled that debugging feature.

The RaceGuard kernel has been in use on various developer workstations (now up to half a dozen) for the last six weeks. Workloads include editing files, compiling & testing code, reading e-mail, surfing the web, playing MP3s[1], and compiling large systems such as the kernel itself (see Section 6). The above are the only compatibility issues found to date, and all of them are addressed by the current implementation.

1. Essential for software development :-)

Table 1: RaceGuard Microbenchmark Results

System Call	Without RaceGuard	With RaceGuard	% Overhead
Stat non-existent file	4.3 microseconds	8.8 microseconds	104%
Open non-existent file	1.5 milliseconds	1.44 milliseconds	-4%
Fork	161 microseconds	183 microseconds	13%

6 Performance Testing

Any run-time security defense will impose performance costs, due to additional run-time checks that it is performing. However, a security enhancement must be efficient enough that these overhead costs are minimal with respect to the defense they provide. Ideally, the cost should be below noticability for the intended user base.

RaceGuard achieves this level of performance. Overhead is only imposed on the run-time cost of a handful of mediated system calls. The cost on each system call is cache insertion or lookup to see if the proposed name is in the RaceGuard cache. Section 6.1 presents microbenchmarks that show the precise overhead imposed on these system calls. Section 6.2 shows macrobenchmarks that measure the imposed overhead on programs that make intensive use of many temporary files.

6.1 Microbenchmarks

Here we measure the marginal overhead of RaceGuard protection on each of the mediated system calls. We measure the overhead with programs that call the affected system call 10,000 times in a tight loop, the test is run 10 times, the lowest and highest are thrown away, and the remainder are averaged. We ran these tests with and without RaceGuard protection, and computed the percent overhead. The performance results are shown in Table 1. Some commentary on the results:.

Stat a non-existent file: measure the overhead to create a RaceGuard cache entry. The marginal overhead is substantial, because the work of the non-RaceGuard case is minimal, while the RaceGuard case is doing some work.

Open non-existent file: measure the overhead to find and clear a RaceGuard cache entry. We do not actually believe there is a speedup due to RaceGuard, and regard this as experimental error, as the differences within the tests exceeded the differences between the tests. We believe this is because the cost of creating a non-existent file is dominated by the state of the file system on disk.

Fork: measure the overhead of copying the RaceGuard cache. This test exhibited significant variances, and so we enhanced measurement to run the test 100 times and took the average. The variance was present in both the RaceGuard and non-RaceGuard tests, so it was not induced by RaceGuard.

Thus there is substantial overhead only in stat'ing non-existent files, and that cost is dwarfed by the cost of creating files. This operation does not represent a large amount of time in a real workload, as we show in our macrobenchmarks in Section 6.2.

6.2 Macrobenchmarks

To stress-test RaceGuard at the macro level, we sought an application that incurred a substantial amount of run time, used many temporary files, and did a lot of forking. Our first selected test is what we call the Khernelstone[1]: the time to build the SRPM of the Linux kernel, which builds the kernel from its 1800 C and assorted assembly source files, several times. Thus this test incorporates several thousand forks and temporary files. The test was run on a single-processor 700 MHz Athlon machine with 128 MB of RAM.

We ran this test four times each with and without RaceGuard. The results showed *very* little variation. The averages of the four runs are shown in Table 2. In all cases (real time, user time, and system time) the overhead due to RaceGuard was always below 0.5%.

Our second macrobenchmark was the Apache web server [3] measured by the Webstone benchmark tool [16]. Webstone simulates various web server workloads, varying the number of concurrent clients and the size of the requests. Webstone measures a variety of factors, including connection latency, and server throughput in terms of both connections per second and bytes per second. Webstone only stresses the web server machine, which in this case was a dual processor 700 MHz Pen-

1. After the venerable Dhrystone integer performance benchmark [20], which in turn is a reference to the Whetstone floating point benchmark.

Table 2: Khernelstone Macrobenchmark, in Seconds

	Real Time	User Time	System Time
Without RaceGuard	10,700	8838	901
With RaceGuard	10,742	8858	904
%Overhead	0.4%	0.2%	0.3%

tium III, 256 KB of cache, 256 MB of RAM, and a 100 Mbit network to the client.

We ran the Webstone test through a range from 25 concurrent clients to 200 concurrent clients. While performance varies substantially across this range, the difference between the RaceGuard and non-RaceGuard cases was always in the noise. WebStone runs each test case three times, and when ever one test run showed one of the cases to be ahead, another test case showed the opposite. In almost all cases, the RaceGuard average is within the non-raceguard average +/- the non-raceguard standard deviation. There are only three cases in which the RaceGuard average is outside the deviation and in each of those three instances we're outside because we do slightly better.[1]

We draw two conclusions from these tests. First, they are apparently not particularly effective at highlighting the costs of RaceGuard. Second, and more importantly, it shows that the performance overhead of RaceGuard is imperceptible for common, heavy workloads.

7 Related Work

The study of temporary file race vulnerabilities is old: Abbott et al [1] and then Bisbey & Hollingsworth [5] described them as a subclass of timing or synchronization flaws. Yet despite the depth of past study of this problem, a practical solution is apparently still needed: temporary file race vulnerabilities were found in core Internet infrastructure tools such as Apache in 2001 [13].

Bishop's seminal paper [6] formally defined the notion of a TOCTTOU (Time Of Check To Time Of Use) error as being two sequential events in which the second depends on the first, and that there is a faulty assumption that results from the first operation will persist to the second operation. Bishop presents a partial solution to TOCTTOU vulnerabilities in the form of a program scanning program to detect some potential TOCTTOU vulnerabilities in C code, but also presents theorems showing that detecting statically TOCTTOU flaws is undecidable.

1. We do not claim that RaceGuard is a performance enhancement, only that performance cost is smaller than experimental error.

Bishop discusses the possibility of a run-time TOCTTOU detector that modifies system call interfaces to track the arguments to system calls, and the association of file descriptors and names, abstractly similar to RaceGuard. Bishop does not elaborate this proposal due to performance concerns. RaceGuard overcomes these performance difficulties by narrowing the scope and duration of the information to be tracked, showing that *near* precise file system race attacks can be detected at run time with very low performance costs.

"Solar Designer" [10] takes a different approach to combating temporary file race vulnerabilities. Rather than attacking the "race" aspect, this enhancement to the Linux kernel attacks the propensity for privileged programs to create and follow links. Two restrictions are imposed: processes will not follow links in directories with the +t ("sticky") bit set unless the owner is trusted (same UID as the process, or owns the directory), and processes are not allowed to create hard links to files that they do not own.

While effective in many cases, this approach unfortunately gets mixed results. Some programs (wrongly) create temporary files in other file systems, e.g. the current working directory. We have also observed real programs that insist on linking to files that they do not own, e.g. the Courier mail server [19] which uses links to optimize the order of mail delivery, and makes links to files with a different UID but a shared GID.

8 Conclusions

Temporary file race vulnerabilities have been a pervasive security problem for over a decade. There are safe methods to create temporary files, but they are not portable, and thus common programs continue to use vulnerable-but-portable temporary file methods such as `mktemp`. RaceGuard protects vulnerable programs against this problem, even if the program insists on using unsafe means, and regardless of whether the program is using an unsafe library, or "rolled their own" unsafe temporary file creation method. We have shown that RaceGuard is effective in stopping attacks, and imposes minimal compatibility and performance overhead. RaceGuard is available as a GPL'd patch to the Linux kernel, and is incorporated into WireX's Immunix server products.

References

[1] R.P. Abbott, J.S. Chin, J.E. Donnelley, W.L. Konigsford, S. Tokubo, and D.A. Webb. Security Analysis and Enhancements of Computer Operating Systems. NSBIR 76-1041, National Bureau of Standards, April 1976.

[2] Assorted. GNU Diffutils. `http://www.gnu.org/software/diffutils/` , June 2000.

[3] Brian Behlendorf, Roy T. Fielding, Rob Hartill, David Robinson, Cliff Skolnick, Randy Terbush, Robert S. Thau, and Andrew Wilson. Apache HTTP Server Project. `http://www.apache.org`.

[4] Brian Berliner, david d 'zoo' zuhn, Jeff Polk, and et al. Concurrent Versions System. `http://www.cyclic.com/`, 1999.

[5] R. Bisbey and D. Hollingsworth. Protection Analysis Project Final Report. Technical Report ISI/RR-78-13, USC/Information Sciences Institute, May 1978. DTICAD A 056816.

[6] M. Bishop and M. Digler. Checking for Race Conditions in File Accesses. *Computing Systems*, 9(2):131–152, Spring 1996. Also available at `http://olympus.cs.ucdavis.edu/ bishop/scriv/index.html`.

[7] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Mike Frantzen, and Jamie Lokier. FormatGuard: Automatic Protection From printf Format String Vulnerabilities. In *USENIX Security Symposium*, Washington, DC, August 2001.

[8] Crispin Cowan, Steve Beattie, Calton Pu, Perry Wagle, and Virgil Gligor. SubDomain: Parsimonious Server Security. In *USENIX 14th Systems Administration Conference (LISA)*, New Orleans, LA, December 2000.

[9] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. In *DARPA Information Survivability Conference and Expo (DISCEX)*, January 2000. Also presented as an invited talk at SANS 2000, March 23-26, 2000, Orlando, FL, `http://schafercorp-ballston.com/discex`.

[10] "Solar Designer". Root Programs and Links. `http://www.openwall.com/linux/`.

[11] Mendel Rosenblum et al. VMWare. `http://www.vmware.com/` , February 1998.

[12] Tomasz Kloczko. Shadow Utilities. `http://freshmeat.net/projects/shadow/` , October 2000.

[13] Greg Kroah-Hartman. Immunix OS Security update for lots of temp file problems. Bugtraq mailing list, `http://www.securityfocus.com/archive/1/155417`, January 10 2001.

[14] MagniComp. RDist. `http://www.magnicomp.com/rdist/` July 1999.

[15] RCS Maintainers. RCS: Revision Control System. `http://www.cs.purdue.edu/homes/trinkle/RCS/` , December 2000.

[16] Mindcraft. WebStone Standard Web Server Benchmark. `http://www.mindcraft.com/webstone/`.

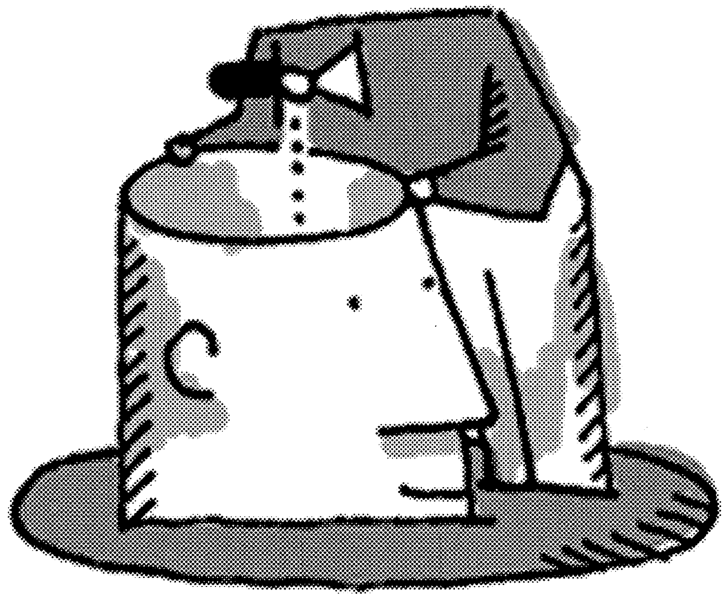
[17] Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9), November 1975.

[18] Securityfocus.com. Vulnerability Search. `http://search.securityfocus.com/search.html`, 1997-2001.

[19] Sam Varshavchik. Courier Mail Transfer Agent. `http://www.courier-mta.org/`, 1999.

[20] Reinhold P. Weicker. Dhrystone: A Synthetic Systems Programming Benchmark. *Communications of the ACM*, 27(10):1013–1030, October 1984.

MANAGING CODE



Session Chair: Trent Jaeger, *IBM T.J. Watson Research Center*

Statically Detecting Likely Buffer Overflow Vulnerabilities

David Larochelle
larochelle@cs.virginia.edu
University of Virginia, Department of Computer Science
David Evans
evans@cs.virginia.edu
University of Virginia, Department of Computer Science

Abstract

Buffer overflow attacks may be today's single most important security threat. This paper presents a new approach to mitigating buffer overflow vulnerabilities by detecting likely vulnerabilities through an analysis of the program source code. Our approach exploits information provided in semantic comments and uses lightweight and efficient static analyses. This paper describes an implementation of our approach that extends the LCLint annotation-assisted static checking tool. Our tool is as fast as a compiler and nearly as easy to use. We present experience using our approach to detect buffer overflow vulnerabilities in two security-sensitive programs.

1. Introduction

Buffer overflow attacks are an important and persistent security problem. Buffer overflows account for approximately half of all security vulnerabilities [CWPBW00, WFBA00]. Richard Pethia of CERT identified buffer overflow attacks as the single most important security problem at a recent software engineering conference [Pethia00]; Brian Snow of the NSA predicted that buffer overflow attacks would still be a problem in twenty years [Snow99].

Programs written in C are particularly susceptible to buffer overflow attacks. Space and performance were more important design considerations for C than safety. Hence, C allows direct pointer manipulations without any bounds checking. The standard C library includes many functions that are unsafe if they are not used carefully. Nevertheless, many security-critical programs are written in C.

Several run-time approaches to mitigating the risks associated with buffer overflows have been proposed. Despite their availability, these techniques are not used widely enough to substantially mitigate the effectiveness of buffer overflow attacks. The next section describes representative run-time approaches and speculates on why they are not more widely used. We propose, instead, to tackle the problem by detecting likely buffer overflow vulnerabilities through a static analysis of program source code. We have implemented a prototype tool that does this by extending LCLint [Evans96]. Our work differs from other work on static detection of buffer overflows in three key ways: (1) we exploit semantic comments added to source code to enable local checking of interprocedural properties; (2) we focus on lightweight static checking techniques that have good performance and scalability characteristics, but sacrifice soundness and completeness; and (3) we introduce loop heuristics, a simple approach for efficiently analyzing many loops found in typical programs.

The next section of this paper provides some background on buffer overflow attacks and previous attempts to mitigate the problem. Section 3 gives an overview of our approach. In Section 4, we report on our experience using our tool on wu-ftpd and BIND, two security-sensitive programs. The following two sections provide some details on how our analysis is done. Section 7 compares our work to related work on buffer overflow detection and static analysis.

2. Buffer Overflow Attacks and Defenses

The simplest buffer overflow attack, *stack smashing* [AlephOne96], overwrites a buffer on the stack to replace the return address. When the function returns, instead of jumping to the return address, control will jump to the address that was placed on the stack by the attacker. This gives the attacker the ability to execute arbitrary code. Programs written in C are particularly susceptible to this type of attack. C provides direct low-level memory access and pointer arithmetic without bounds checking. Worse, the standard C library provides unsafe functions (such as `gets`) that write an unbounded amount of user input into a fixed size buffer without any bounds checking [ISO99]. Buffers stored on the stack are often passed to these functions. To exploit such vulnerabilities, an attacker merely has to enter an input larger than the size of the buffer and encode an attack program binary in that input. The Internet Worm of 1988 [Spafford88, RE89] exploited this type of buffer overflow vulnerability in fingerd. More sophisticated buffer overflow attacks may exploit unsafe buffer usage on the heap. This is harder, since most programs do not jump to addresses loaded from the heap or to code that is stored in the heap.

Several run-time solutions to buffer overflow attacks have been proposed. StackGuard [CPMH+98] is a compiler that generates binaries that incorporate code designed to prevent stack smashing attacks. It places a special value on the stack next to the return address, and checks that it has not been tampered with before jumping. Baratloo, Singh and Tsai describe two run-time approaches: one replaces unsafe library functions with safe implementations; the other modifies executables to perform sanity checking of return addresses on the stack before they are used [BST00].

Software fault isolation (SFI) is a technique that inserts bit mask instructions before memory operations to prevent access of out-of-range memory [WLAG93]. This alone does not offer much protection against typical buffer overflow attacks since it would not prevent a program from writing to the stack address where the return value is stored. Generalizations of SFI can insert more expressive checking around potentially dangerous operations to restrict the behavior of programs more generally. Examples include Janus, which observes and mediates behavior by monitoring system calls [GWTB96]; Naccio [ET99, Evans00a] and PSLang/PoET [ES99, ES00] which transform object programs according to a safety policy; and Generic Software Wrappers [FBF99] which wraps system calls with security checking code.

Buffer overflow attacks can be made more difficult by modifications to the operating system that put code and data in separate memory segments, where the code segment is read-only and instructions cannot be executed from the data segment. This does not eliminate the buffer overflow problem, however, since an attacker can still overwrite an address stored on the stack to make the program jump to any point in the code segment. For programs that use shared libraries, it is often possible for an attacker to jump to an address in the code segment that can be used maliciously (e.g., a call to `system`). Developers decided against using this approach in the Linux kernel since it did not solve the real problem and it would prevent legitimate uses of self-modifying code [Torvalds98, Coolbaugh99].

Despite the availability of these and other run-time approaches, buffer overflow attacks remain a persistent problem. Much of this may be due to lack of awareness of the severity of the problem and the availability of practical solutions. Nevertheless, there are legitimate reasons why the run-time solutions are unacceptable in some environments. Run-time solutions always incur some performance penalty (StackGuard reports performance overhead of up to 40% [CBDP+99]). The other problem with run-time solutions is that while they may be able to detect or prevent a buffer overflow attack, they effectively turn it into a denial-of-service attack. Upon detecting a buffer overflow, there is often no way to recover other than terminating execution.

Static checking overcomes these problems by detecting likely vulnerabilities before deployment. Detecting buffer overflow vulnerabilities by analyzing code in general is an undecidable problem.[1] Nevertheless, it is possible to produce useful results using static analysis. Rather than attempting to verify that a program has no buffer overflow vulnerabilities, we wish to have reasonable confidence of detecting a high fraction of likely buffer overflow vulnerabilities. We are willing to accept a solution that is both unsound and incomplete. This means that our checker will sometimes generate false warnings and sometimes miss real problems. Our goal is to produce a tool that produces useful results for real programs with a reasonable effort. The next section describes our approach. We compare our work with other static approaches to detecting buffer overflow vulnerabilities in Section 7.

3. Approach

Our static analysis tool is built upon LCLint [EGHT94, Evans96, Evans00b], an annotation-assisted lightweight static checking tool. Examples of problems detected by LCLint include violations of information hiding, inconsistent modifications of caller-visible state or uses of global variables, misuses of possibly NULL references, uses of dead storage, memory leaks and problems with parameters aliasing. LCLint is actually

[1] We can trivially reduce the halting problem to the buffer overflow detection problem by inserting code that causes a buffer overflow before all halt instructions.

used by working programmers, especially in the open source development community [Orcero00, PG00].

Our approach is to exploit semantic comments (henceforth called *annotations*) that are added to source code and standard libraries. Annotations describe programmer assumptions and intents. They are treated as regular C comments by the compiler, but recognized as syntactic entities by LCLint using the `@` following the `/*` to identify a semantic comment. For example, the annotation `/*@notnull@*/` can be used syntactically like a type qualifier. In a parameter declaration, it indicates that the value passed for this parameter may not be NULL. Although annotations can be used on any declaration, for this discussion we will focus exclusively on function and parameter declarations. We can also use annotations similarly in declarations of global and local variables, types and type fields.

Annotations constrain the possible values a reference can contain either before or after a function call. For example, the `/*@notnull@*/` annotation places a constraint on the parameter value before the function body is entered. When LCLint checks the function body, it assumes the initial value of the parameter is not NULL. When LCLint checks a call site, it reports a warning unless it can determine that the value passed as the corresponding parameter is never NULL.

Prior to this work, all annotations supported by LCLint classified references as being in one of a small number of possible states. For example, the annotation `/*@null@*/` indicated that a reference may be NULL, and the annotation `/*@notnull@*/` indicated that a reference is not NULL. In order to do useful checking of buffer overflow vulnerabilities, we need annotations that are more expressive. We are concerned with how much memory has been allocated for a buffer, something that cannot be adequately modeled using a finite number of states. Hence, we need to extend LCLint to support a more general annotation language. The annotations are more expressive, but still within the spirit of simple semantic comments added to programs.

The new annotations allow programmers to explicitly state function preconditions and postconditions using requires and ensures clauses.[2] We can use these clauses to describe assumptions about buffers that are passed to functions and constrain the state of buffers when functions return. For the analyses described in this paper, four kinds of assumptions and constraints are used: minSet, maxSet, minRead and maxRead.[3]

When used in a requires clause, the minSet and maxSet annotations describe assumptions about the lowest and highest indices of a buffer that may be safely used as an lvalue (e.g., on the left-hand side of an assignment). For example, consider a function with an array parameter `a` and an integer parameter `i` that has a precondition requires maxSet(a) >= i. The analysis assumes that at the beginning of the function body, `a[i]` may be used as an lvalue. If `a[i+1]` were used before any modifications to the value of `a` or `i`, LCLint would generate a warning since the function preconditions are not sufficient to guarantee that `a[i+1]` can be used safely as an lvalue. Arrays in C start with index `0`, so the declaration

```
char buf[MAXSIZE]
```

generates the constraints

maxSet(buf) = MAXSIZE – 1 and
minSet(buf) = 0.

Similarly, the minRead and maxRead constraints indicate the minimum and maximum indices of a buffer that may be read safely. The value of maxRead for a given buffer is always less than or equal to the value of maxSet. In cases where there are elements of the buffer have not yet been initialized, the value of maxRead may be lower than the value of maxSet.

At a call site, LCLint checks that the preconditions implied by the requires clause of the called function are satisfied before the call. Hence, for the requires maxSet(a) >= i example, it would issue a warning if it cannot determine that the array passed as `a` is allocated to hold at least as many elements as the value passed as `i`. If minSet or maxSet is used in an ensures clause, it indicates the state of a buffer after the function returns. Checking at the call site proceeds by assuming the postconditions are true after the call returns.

For checking, we use an annotated version of the standard library headers. For example, the function `strcpy` is annotated as[4]:

[2] The original Larch C interface language LCL [GH93], on which LCLint's annotation language was based, did include a notion of general preconditions and postconditions specified by requires and ensures clauses.

[3] LCLint also supports a nullterminated annotation that denotes storage that is terminated by the null character. Many C library functions require null-terminated strings, and can produce buffer overflow vulnerabilities if they are passed a string that is not properly null-terminated. We do not cover the nullterminated annotation and related checking in this paper. For information on it, see [LHSS00].

[4] The standard library specification of `strcpy` also includes other LCLint annotations: a modifies clause that indicates that the only thing that may be modified by `strcpy` is the storage referenced by `s1`, an out annotation on `s1` to indicate that it

```
char *strcpy (char *s1, const char *s2)
 /*@requires maxSet(s1) >= maxRead(s2)@*/
 /*@ensures maxRead(s1) == maxRead(s2)
     /\ result == s1@*/;
```

The requires clause specifies the precondition that the buffer `s1` is allocated to hold at least as many characters as are readable in the buffer `s2` (that is, the number of characters up to and including its null terminator). The postcondition reflects the behavior of `strcpy` – it copies the string pointed to by `s2` into the buffer `s1`, and returns that buffer. In ensures clauses, we use the result keyword to denote the value returned by the function.

Many buffer overflows result from using library functions such as `strcpy` in unsafe ways. By annotating the standard library, many buffer overflow vulnerabilities can be detected even before adding any annotations to the target program. Selected annotated standard library functions are shown in Appendix A.

4. Experience

In order to test our approach, we used our tool on wu-ftpd, a popular open source ftp server, and BIND (Berkeley Internet Name Domain), a set of domain name tools and libraries that is considered the reference implementation of DNS. This section describes the process of running LCLint on these applications, and illustrates how our checking detected both known and unknown buffer overflow vulnerabilities in each application.

4.1 wu-ftpd

We analyzed wu-ftp-2.5.0[5], a version with known security vulnerabilities.

Running LCLint is similar to running a compiler. It is typically run from the command line by listing the source code files to check, along with flags that set checking parameters and control which classes of warnings are reported. It takes just over a minute for LCLint to analyze all 17 000 lines of wu-ftpd. Running LCLint on the entire unmodified source code for wu-ftpd without adding any annotations resulted in 243 warnings related to buffer overflow checking.

Consider a representative message[6]:

```
ftpd.c:1112:2: Possible out-of-bounds store. Unable to
  resolve constraint:
    maxRead ((entry->arg[0] @ ftpd.c:1112:23)) <= (1023)
  needed to satisfy precondition:
    requires maxSet ((ls_short @ ftpd.c:1112:14))
          >= maxRead ((entry->arg[0] @ ftpd.c:1112:23))
  derived from strcpy precondition:
    requires maxSet (<param 1>) >= maxRead (<param 2>)
```

Relevant code fragments are shown below with line 1112 in bold:

```
char  ls_short[1024];
...
extern struct aclmember *
getaclentry(char *keyword,
            struct aclmember **next);
...
int main(int argc, char **argv,
         char **envp)
{
   ...
   entry = (struct aclmember *) NULL;
   if (getaclentry("ls_short", &entry)
        && entry->arg[0]
        && (int)strlen(entry->arg[0]) > 0)
      {
        strcpy(ls_short,entry->arg[0]);
        ...
```

This code is part of the initialization code that reads configuration files. Several buffer overflow vulnerabilities were found in the wu-ftpd initialization code. Although this vulnerability is not likely to be exploited, it can cause security holes if an untrustworthy user is able to alter configuration files.

The warning message indicates that a possible out-of-bounds store was detected on line 1112 and contains information about the constraint LCLint was unable to resolve. The warning results from the function call to `strcpy`. LCLint generates a precondition constraint corresponding to the `strcpy` requires clause

need not point to defined storage when `strcpy` is called, a unique annotation on `s1` to indicate that it may not alias the same storage as `s2`, and a returned annotation on `s1` to indicate that the returned pointer references the same storage as `s1`. For clarity, the examples in this paper show only the annotations directly relevant to detecting buffer overflow vulnerabilities. For more information on other LCLint annotations, see [Evans96, Evans00c].

[5] The source code for wu-ftpd is available from http://www.wu-ftpd.org. We analyzed the version in ftp://ftp.wu-ftpd.org/pub/wu-ftpd-attic/wu-ftpd-2.5.0.tar.gz. We configured wu-ftpd using the default configuration for FreeBSD systems. Since LCLint performs most of its analyses on code that has been pre-processed, our analysis did not examine platform-specific code in wu-ftpd for platforms other than FreeBSD.

[6] For our prototype implementation, we have not yet attempted to produce messages that can easily be interpreted by typical programmers. Instead, we generate error messages that reveal information useful to the LCLint developers. Generating good error messages is a challenging problem; we plan to devote more effort to this before publicly releasing our tool.

maxSet(s1) >= maxRead(s2) by substituting the actual parameters:

maxSet (ls_short @ ftpd.c:1112:14)
>= maxRead (entry->arg[0] @ ftpd.c:1112:23).

Note that the locations of the expressions passed as actual parameters are recorded in the constraint. Since values of expressions may change through the code, it is important that constraints identify values at particular program points.

The global variable `ls_short` was declared as an array of 1024 characters. Hence, LCLint determines maxSet (ls_short) is 1023. After the call to `getaclentry`, the local `entry->arg[0]` points to a string of arbitrary length read from the configuration file. Because there are no annotations on the `getaclentry` function, LCLint does not assume anything about its behavior. In particular, the value of maxRead (entry->arg[0]) is unknown. LCLint reports a possible buffer misuse, since the constraint derived from the `strcpy` requires clause may not be satisfied if the value of maxRead (entry->arg[0]) is greater than 1023.

To fix this problem, we modified the code to handle these values safely by using `strncpy`. Since `ls_short` is a fixed size buffer, a simple change to use `strncpy` and store a null character at the end of the buffer is sufficient to ensure that the code is safe.[7]

In other cases, eliminating a vulnerability involved both changing the code and adding annotations. For example, LCLint generated a warning for a call to `strcpy` in the function `acl_getlimit`:

```
int acl_getlimit(char *class,
                  char *msgpathbuf) {
 int limit;
 struct aclmember *entry = NULL;

 if (msgpathbuf) *msgpathbuf = '\0';
 while (getaclentry("limit", &entry)) {
   ...
   if (!strcasecmp(class, entry->arg[0]))
   {
     ...
     if (entry->arg[3]
         && msgpathbuf != NULL)
       strcpy(msgpathbuf, entry->arg[3]);
     ...
```

If the size of `msgputhbuf` is less than the length of the string in `entry->arg[3]`, there is a buffer overflow. To fix this we replaced the `strcpy` call with a safe call to `strncpy`:

```
 strncpy(msgpathbuf, entry->arg[3], 199);
 msgpathbuf[199] = '\0';
```

and added a requires clause to the function declaration:

```
 /*@requires maxSet(msgpathbuf) >= 199@*/
```

The requires clause documents an assumption (that may be incorrect) about the size of the buffer passed to `acl_getlimit`. Because of the constraints denoted by the requires clauses, LCLint does not report a warning for the call to `strncpy`.

When call sites are checked, LCLint produces a warning if it is unable to determine that this requires clause is satisfied. Originally, we had modified the function `acl_getlimit` by adding the precondition maxSet (msgpathbuf) >= 1023. After adding this precondition, LCLint produced a warning for a call site that passed a 200-byte buffer to `acl_getlimit`. Hence, we replaced the requires clause with the stronger constraint and used 199 as the parameter to `strncpy`.

This vulnerability was still present in the current version of wu-ftpd. We contacted the wu-ftpd developers who acknowledged the bug but did not consider it security critical since the string in question is read from a local file not user input [Luckin01, Lundberg01].

In addition to the previously unreported buffer overflows in the initialization code, LCLint detected a known buffer overflow in wu-ftpd. The buffer overflow occurs in the function `do_elem` shown below, which passes a global buffer and its parameters to the library function `strcat`. The function `mapping_chdir` calls `do_elem` with a value entered by the remote user as its parameter. Because wu-ftpd fails to perform sufficient bounds checking, a remote user is able to exploit this vulnerability to overflow the buffer by carefully creating a series of directories and executing the cd command.[8]

```
char mapped_path [200];
...
void do_elem(char *dir) {
   ...
   if (!(mapped_path[0] == '/'
       && mapped_path[1] == '\0'))
     strcat (mapped_path, "/");
   strcat (mapped_path, dir);
}
```

[7] Because `strncpy` does not guarantee null termination, it is necessary to explicitly put a null character at the end of the buffer.

[8] Advisories for this vulnerability can be found at http://www.cert.org/advisories/CA-1999-13.html and ftp://www.auscert.org.au/security/advisory/AA-1999.01.wu-ftpd.mapping_chdir.vul.

LCLint generates warnings for the unsafe calls to `strcat`. This was fixed in latter versions of wu-ftpd by calling `strncat` instead of `strcat`.

Because of the limitations of static checking, LCLint sometimes generates spurious error messages. If the user believes the code is correct, annotations can be added to precisely suppress spurious messages.

Often the code was too complex for LCLint to analyze correctly. For example, LCLint reports a spurious warning for this code fragment since it cannot determine that `((1.0*j*rand()) / (RAND_MAX + 1.0))` always produces a value between `1` and `j`:

```
i = passive_port_max
    - passive_port_min + 1;
port_array = calloc (i, sizeof (int));
for (i = 3; … && (i > 0); i--) {
  for (j = passive_port_max
          - passive_port_min + 1;
      … && (j > 0); j--) {
    k = (int) ((1.0 * j * rand())
              / (RAND_MAX + 1.0));
    pasv_port_array [j-1]
      = port_array [k];
```

Determining that the `port_array[k]` reference is safe would require far deeper analysis and more precise specifications than is feasible within a lightweight static checking tool.

Detecting buffer overflows with LCLint is an iterative process. Many of the constraints we found involved functions that are potentially unsafe. We added function preconditions to satisfy these constraints where possible. In certain cases, the code was too convoluted for LCLint to determine that our preconditions satisfied the constraints. After convincing ourselves the code was correct, we added annotations to suppress the spurious warnings.

Before any annotations were added, running LCLint on wu-ftpd resulted in 243 warnings each corresponding to an unresolved constraint. We added 22 annotations to the source code through an iterative process similar to the examples described above. Nearly all of the annotations were used to indicate preconditions constraining the value of maxSet for function parameters.

After adding these annotations and modifying the code, running LCLint produced 143 warnings. Of these, 88 reported unresolved constraints involving maxSet. While we believe the remaining warnings did not indicate bugs in wu-ftpd, LCLint's analyses were not sufficiently powerful to determine the code was safe. Although this is a higher number of spurious warnings than we would like, most of the spurious warnings can be quickly understood and suppressed by the user. The source code contains 225 calls to the potentially buffer overflowing functions `strcat`, `strcpy`, `strncat`, `strncpy`, `fgets` and `gets`. Only 18 of the unresolved warnings resulted from calls to these functions. Hence, LCLint is able to determine that 92% of these calls are safe automatically. The other warnings all dealt with classes of problems that could not be detected through simple lexical techniques.

4.2 BIND

BIND is a key component of the Internet infrastructure. Recently, the Wall Street Journal identified buffer overflow vulnerabilities in BIND as a critical threat to the Internet [WSJ01]. We focus on named, the DNS sever portion of BIND, in this case study. We analyzed BIND version 8.2.2p7[9], a version with known bugs. BIND is larger and more complex than wu-ftpd. The name server portion of BIND, named, contains approximately 47 000 lines of C including shared libraries. LCLint took less than three and a half minutes to check all of the named code.

We limited our analysis to a subset of named because of the time required for human analysis. We focused on three files: ns_req.c and two library files that contain functions which are called extensively by ns_req.c: ns_name.c and ns_sign.c. These files contain slightly more than 3 000 lines of code.

BIND makes extensive use of functions in its internal library rather than C library functions. In order to accurately analyze individual files, we needed to annotate the library header files. The most accurate way to annotate the library would be to iteratively run LCLint on the library and add annotations. However, the library was extremely large and contains deeply nested call chains. To avoid the human analysis this would require, we added annotations to some of the library functions without annotating all the dependent functions. In many cases, we were able to guess preconditions by using comments or the names of function parameters. For example, several functions took a pointer parameter (p) and another parameter encoding it size (psize), from which we inferred a precondition MaxSet(p) >= (psize – 1). After annotating selected BIND library functions, we were able to check the chosen files without needing to fully annotate all of BIND.

LCLint produces warnings for a series of unguarded buffer writes in the function `req_query`. The code in

[9] The source code is available at
ftp://ftp.isc.org/isc/bind/src/8.2.2-P7/bind-src.tar.gz

question is called in response to a specific type of query which requests information concerning the domain name server version. BIND appends a response to the buffer containing the query that includes a global string read from a configuration file. If the default configuration is used, the code is safe because this function is only called with buffers that are large enough to store the response. However, the restrictions on the safe use of this function are not obvious and could easily be overlooked by someone modifying the code. Additionally, it is possible that an administrator could reconfigure BIND to use a value for the server version string large enough to make the code unsafe. The BIND developers agreed that a bounds check should be inserted to eliminate this risk [Andrews01].

BIND uses extensive run time bounds checking. This type of defensive programming is important for writing secure programs, but does not guarantee that a program is secure. LCLint detected a known buffer overflow in a function that used run time checking but specified buffer sizes incorrectly.[10]

The function `ns_req` examines a DNS query and generates a response. As part of its message processing, it looks for a signature and signs its response with the function `ns_sign`. LCLint reported that it was unable to satisfy a precondition for `ns_sign` that requires the size of the message buffer be accurately described by a size parameter. This precondition was added when we initially annotated the shared library. A careful hand analysis of this function reveals that to due to careless modification of variables denoting buffer length, it is possible for the buffer length to be specified incorrectly if the message contains a signature but a valid key is not found. This buffer overflow vulnerability was introduced when a digital signature feature was added to BIND (ironically to increase security). Static analysis tools can be used to quickly alert programmers to assumptions that are broken by incremental code changes.

Based on our case studies, we believe that LCLint is a useful tool for improving the security of programs. It does not detect all possible buffer overflow vulnerabilities, and it can generate spurious warnings. In practice, however, it provides programmers concerned about security vulnerabilities with useful assistance, even for large, complex programs. In addition to aiding in the detection of exploitable buffer overflows, the process of adding annotations to code encourages a disciplined style of programming and produces programs that include reliable and precise documentation.

[10] An advisory for this vulnerability can be found at http://lwn.net/2001/0201/a/covert-bind.php3.

5. Implementation

Our analysis is implemented by combining traditional compiler data flow analyses with constraint generation and resolution. Programs are analyzed at the function level; all interprocedural analyses are done using the information contained in annotations.

We support four types of constraints corresponding to the annotations introduced in Section 2: maxSet, minSet, maxRead, and minRead. Constraints can also contain constants and variables and allow the arithmetic operations: + and -. Terms in constraints can refer to any C expression, although our analysis will not be able to evaluate some C expressions statically.

The full constraint grammar is:

constraint ⇒ (requires | ensures)
constraintExpression relOp constraintExpression
relationalOp ⇒ == | > | >= | < | <=
constraintExpression ⇒
  constraintExpression binaryOp constraintExpresion
  | *unaryOp* (*constraintExpression*)
  | *term*
binaryOp ⇒ + | -
unaryOp ⇒ maxSet | maxRead | minSet | minRead
term ⇒ *variable* | C *expression* | *literal* | result

Source-code annotations allow arbitrary constraints (as defined by our constraint grammar) to be specified as the preconditions and postconditions of functions. Constraints can be conjoined (using $\wedge$), but there is no support for disjunction. All variables used in constraints have an associated location. Since the value stored by a variable may change in the function body, it is important that the constraint resolver can distinguish the value at different points in the program execution.

Constraints are generated at the expression level and stored in the corresponding node in the parse tree. Constraint resolution is integrated with the checking by resolving constraints at the statement level as checking traverses up the parse tree. Although this limits the power of our analysis, it ensures that it will be fast and simple. The remainder of this section describes briefly how constraints are represented, generated and resolved.

Constraints are generated for C statements by traversing the parse tree and generating constraints for each subexpression. We determine constraints for a statement by conjoining the constraints of its subexpressions. This assumes subexpressions cannot change state that is used by other subexpressions of the same expression. The semantics of C make this a valid assumption for nearly all expressions – it is undefined behavior in C for two subexpressions not separated by a sequence point to read and write the same data. Since LCLint detects and warns about this type of undefined behavior, it is reasonable for the buffer overflow checking to rely on this assumption. A few C expressions do have intermediate sequence points (such as the comma operator which specifies that the left operand is always evaluated first) and cannot be analyzed correctly by our simplified assumptions. In practice, this has not been a serious limitation for our analysis.

Constraints are resolved at the statement level in the parse tree and above using axiomatic semantics techniques. Our analysis attempts to resolve constraints using postconditions of earlier statements and function preconditions. To aid in constraint resolution, we simplify constraints using standard algebraic techniques such as combining constants and substituting terms. We also use constraint-specific simplification rules such as maxSet(*ptr* + *i*) = maxSet(*ptr*) - *i*. We have similar rules for maxRead, minSet, and minRead.

Constraints for statement lists are produced using normal axiomatic semantics rules and simple logic to combine the constraints of individual statements. For example, the code fragment

```
1       t++;
2       *t = 'x';
3       t++;
```

leads to the constraints:

requires maxSet(t @ 1:1) >= 1,
ensures maxRead(t @ 3:4) >= -1 and
ensures (t @ 3:4) = (t @ 1:1) + 2.

The assignment to `*t` on line 2 produces the constraint requires maxSet(t @ 2:2) >= 0. The increment on line 1 produces the constraint ensures (t@1:4) = (t@1:1) + 1. The increment constraint is substituted into the maxSet constraint to produce requires maxSet (t@1:1 + 1) >= 0. Using the constraint-specific simplification rule, this simplifies to requires maxSet (t@1:1) - 1 >= 0 which further simplifies to requires maxSet(t @ 1:1) >= 1.

6. Control Flow

Statements involving control flow such as `while` and `for` loops and `if` statements, require more complex analysis than simple statement lists. For `if` statements and loops, the predicate often provides a guard that makes a possibly unsafe operation safe. In order to analyze such constructs well, LCLint must take into account the value of the predicate on different code paths. For each predicate, LCLint generates three lists of postcondition constraints: those that hold regardless of the truth value of the predicate, those that hold when the predicate evaluates to true, and those that hold when the predicate evaluates to false.

To analyze an `if` statement, we develop branch specific guards based on our analysis of the predicate and use these guards to resolve constraints within the body. For example, in the statement

```
if (sizeof (s1) > strlen (s2))
   strcpy(s1, s2);
```

if `s1` is a fixed-size array, `sizeof (s1)` will be equal to maxSet(s1) + 1. Thus the if predicate allows LCLint to determine that the constraint maxSet(s1) >= maxRead(s2) holds on the true branch. Based on this constraint LCLint determines that the call to `strcpy` is safe.

Looping constructs present additional problems. Previous versions of LCLint made a gross simplification of loop behavior: all `for` and `while` loops in the program were analyzed as though the body executed either zero or one times. Although this is clearly a ridiculous assumption, it worked surprisingly well for the types of analyses done by LCLint. For the buffer overflow analyses, this simplified view of loop semantics does not provide satisfactory results – to determine whether `buf[i]` is a potential buffer overflow, we need to know the range of values `i` may represent. Analyzing the loop as though its body executed only once would not provide enough information about the possible values of `i`.

In a typical program verifier, loops are handled by requiring programmers to provide loop invariants. Despite considerable effort [Wegbreit75, Cousot77, Collins88, IS97, DLNS98, SI98], no one has yet been able to produce tools that generate suitable loop invariants automatically. Some promising work has been done towards discovering likely invariants by executing programs [ECGN99], but these techniques require well-constructed test suites and many problems remain before this could be used to produce the kinds of loop invariants we need. Typical programmers are not able or willing to annotate their code with loop invariants, so for LCLint to be effective we needed a method for handling loops that produces better results than our previous gross simplification method, but did not require expensive analyses or programmer-supplied loop invariants.

Our solution is to take advantage of the idioms used by typical C programmers. Rather than attempt to handle all possible loops in a general way, we observe that a large fraction of the loops in most C programs are written in a stylized and structured way. Hence, we can develop heuristics for identifying and analyzing loops that match certain common idioms. When a loop matches a known idiom, corresponding heuristics can be used to guess how many times the loop body will execute. This information is used to add additional preconditions to the loop body that constrain the values of variables inside the loop.

To further simplify the analysis, we assume that any buffer overflow that occurs in the loop will be apparent in either the first or last iterations. This is a reasonable assumption in almost all cases, since it would be quite rare for a program to contain a loop where the extreme values of loop variables were not on the first and last iterations. This allows simpler and more efficient loop checking. To analyze the first iteration of the loop, we treat the loop as an `if` statement and use the techniques described above. To analyze the last iteration we use a series of heuristics to determine the number of loop iterations and generate additional constraints based on this analysis.

An example loop heuristic analyzes loops of the form

```
for (index = 0; expr; index++) body
```

where the *body* and *expr* do not modify the *index* variable and *body* does not contain a statement (e.g., a `break`) that could interfere with normal loop execution. Analyses performed by the original LCLint are used to aid loop heuristic pattern matching. For example, we use LCLint's modification analyses to determine that the loop body does not modify the index variable.

For a loop that matches this idiom, it is reasonable to assume that the number of iterations can be determined solely from the loop predicate. As with `if` statements, we generate three lists of postcondition constraints for the loop test. We determine the terminating condition of the loop by examining the list of postcondition constraints that apply specifically to the true branch. Within these constraints, we look for constraints of the form *index* <= *e*. For each of these constraints, we search the increment part of the loop header for constraints matching the form *index* = *index* + 1. If we find a constraint of this form, we assume the loop runs for *e* iterations.

Of course, many loops that match this heuristic will not execute for *e* iterations. Changes to global state or other variables in the loop body could affect the value of *e*. Hence, our analysis is not sound or complete. For the programs we have tried so far, we have found this heuristic works correctly.

Abstract syntax trees for loops are converted to a canonical form to increase their chances of matching a known heuristic. After canonicalization, this loop pattern matches a surprisingly high number of cases. For example, in the loop

```
for (i = 0; buffer[i]; i++) body
```

the postconditions of the loop predicate when the body executes would include the constraint ensures i < maxRead(buffer). This would match the pattern so LCLint could determine that the loop executes for maxRead(buffer) iterations.

Several other heuristics are used to match other common loop idioms used in C programs. We can generalize the first heuristic to cases where the initial index value is not known. If LCLint can calculate a reasonable upper bound on the number of iterations (for example, if we can determine that the initial value of the index is always non-negative), it can determine an upper bound on the number of loop iterations. This can generate false positives if LCLint overestimates the actual number of loop iterations, but usually gives a good enough approximation for our purposes.

Another heuristic recognizes a common loop form in which a loop increments and tests a pointer. Typically, these loops match the pattern:

```
for (init; *buf; buf++)
```

A heuristic detects this loop form and assumes that loop executes for maxRead(buf) iterations.

After estimating the number of loop iterations, we use a series of heuristics to generate reasonable constraints for the last iteration. To do this, we calculate the value of each variable in the last iteration. If a variable is incremented in the loop, we estimate that in the last iteration the variable is the sum of the number of loop iterations and the value of the variable in the first iteration. For the loop to be safe, all loop preconditions involving the variable must be satisfied for the values of the variable in both the first and last iterations. This heuristic gives satisfactory results in many cases.

Our heuristics were initially developed based on our analysis of wu-ftpd. We found that our heuristics were effective for BIND also. To handle BIND, a few additional heuristics were added. In particular, BIND frequently used comparisons of pointer addresses to ensure a memory accesses is safe. Without an appropriate heuristic, LCLint generated spurious warnings for these cases. We added appropriate heuristics to handle these situations correctly. While we expect experience with additional programs would lead to the addition of new loop heuristics, it is encouraging that only a few additional heuristics were needed to analyze BIND.

Although no collection of loop heuristics will be able to correctly analyze all loops in C programs, our experience so far indicates that a small number of loop heuristics can be used to correctly analyze most loops in typical C programs. This is not as surprising as it might seem – most programmers learn to code loops from reading examples in standard texts or other people's code. A few simple loop idioms are sufficient for programming many computations.

7. Related Work

In Section 2, we described run-time approaches to the buffer overflow problem. In this section, we compare our work to other work on static analysis.

It is possible to find some program flaws using lexical analysis alone. Unix grep is often used to perform a crude analysis by searching for potentially unsafe library function calls. ITS4 is a lexical analysis tool that searches for security problems using a database of potentially dangerous constructs [VBKM00]. Lexical analysis techniques are fast and simple, but their power is very limited since they do not take into account the syntax or semantics of the program.

More precise checking requires a deeper analysis of the program. Our work builds upon considerable work on constraint-based analysis techniques. We do not attempt to summarize foundational work here. For a summary see [Aiken99].

Proof-carrying code [NL 96, Necula97] is a technique where a proof is distributed with an executable and a verifier checks the proof guarantees the executable has certain properties. Proof-carrying code has been used to enforce safety policies constraining readable and writeable memory locations. Automatic construction of proofs of memory safety for programs written in an unsafe language, however, is beyond current capabilities.

Wagner, et al. have developed a system to statically detect buffer overflows in C [WFBA00, Wagner00]. They used their tool effectively to find both known and unknown buffer overflow vulnerabilities in a version of sendmail. Their approach formulates the problem as an integer range analysis problem by treating C strings as an abstract type accessed through library functions and modeling pointers as integer ranges for allocated size and length. A consequence of modeling strings as an abstract data type is that buffer overflows involving non-character buffers cannot be detected. Their system generates constraints similar to those generated by LCLint for operations involving strings. These constraints are not generated from annotations, but constraints for standard library functions are built in to the tool. Flow insensitive analysis is used to resolve the constraints. Without the localization provided by annotations, it was believed that flow sensitive analyses would not scale well enough to handle real programs. Flow insensitive analysis is less accurate and does not allow special handling of loops or `if` statements.

Dor, Rodeh and Sagiv have developed a system that detects unsafe string operations in C programs [DRS01]. Their system performs a source-to-source transformation that instruments a program with additional variables that describe string attributes and contains `assert` statements that check for unsafe string operations. The instrumented program is then analyzed statically using integer analysis to determine possible assertion failures. This approach can handle many complex properties such as overlapping pointers. However, in the worst case the number of variables in the instrumented program is quadratic in the number of variables in the original program. To date, it has only been used on small example programs.

A few tools have been developed to detect array bounds errors in languages other than C. John McHugh developed a verification system that detects array bounds errors in the Gypsy language [McHugh84]. Extended Static Checking uses an automatic theorem-prover to detect array index bounds errors in Modula-3 and Java [DLNS98]. Extended Static Checking uses information in annotations to assist checking. Detecting array bounds errors in C programs is harder than for Modula-3 or Java, since those languages do not provide pointer arithmetic.

8. Conclusions

We have presented a lightweight static analysis tool for detecting buffer overflow vulnerabilities. It is neither sound nor complete; hence, it misses some vulnerabilities and produces some spurious warnings. Despite this, our experience so far indicates that it is useful. We were able to find both known and previously unknown buffer overflow vulnerabilities in wu-ftpd and BIND with a reasonable amount of effort using our approach. Further, the process of adding annotations is a constructive and useful step for understanding of a program and improving its maintainability.

We believe it is realistic (albeit perhaps optimistic) to believe programmers would be willing to add annotations to their programs if they are used to efficiently and clearly detect likely buffer overflow vulnerabilities (and other bugs) in their programs. An informal sampling of tens of thousands of emails received from LCLint users indicates that about one quarter of LCLint users add the annotations supported by previously released versions of LCLint to their programs. Perhaps half of those use annotations in sophisticated ways (and occasionally in ways the authors never imagined). Although the annotations required for effectively detecting buffer overflow vulnerabilities are somewhat more complicated, they are only an incremental step beyond previous annotations. In most cases, and certainly for security-sensitive programs, the benefits of doing so should far outweigh the effort required.

These techniques, and static checking in general, will not provide the complete solution to the buffer overflow problem. We are optimistic, though, that this work represents a step towards that goal.

Availability

LCLint source code and binaries for several platforms are available from http://lclint.cs.virginia.edu.

Acknowledgements

We would like to thank the NASA Langley Research Center for supporting this work. David Evans is also supported by an NSF CAREER Award. We thank John Knight, John McHugh, Chenxi Wang, Joel Winstead and the anonymous reviewers for their helpful and insightful comments.

References

[Aiken99] Alexander Aiken. *Introduction to Set Constraint-Based Program Analysis*. Science of Computer Programming, Volume 35, Numbers 2-3. November 1999.

[AlephOne96] Aleph One. *Smashing the Stack for Fun and Profit*. BugTraq Archives. http://immunix.org/StackGuard/profit.html.

[Andrews01] Mark Andrews. Personal communication, May 2001.

[BST00] Arash Baratloo, Navjot Singh and Timothy Tsai. *Transparent Run-Time Defense Against Stack-Smashing Attacks*. 9th USENIX Security Symposium, August 2000.

[Collins88] William J. Collins. *The Trouble with For-Loop Invariants*. 19th SIGCSE Technical Symposium on Computer Science Education, February 1988.

[Coolbaugh99] Liz Coolbaugh. *Buffer Overflow Protection from Kernel Patches*. Linux Weekly News, http://lwn.net/1999/1230/security.php3.

[Cousot77] Patrick Cousot and Radhia Cousot. *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*. Fourth ACM Symposium on Principles of Programming Languages, January 1977.

[CPMH+98] Crispin Cowan, Calton Pu, David Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle and Qian Zhang. *Automatic Detection and Prevention of Buffer-Overflow Attacks*. 7th USENIX Security Symposium, January 1998.

[CBDP+99] Crispin Cowan, Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle and Erik Walthinsen. *Protecting Systems from Stack Smashing Attacks with StackGuard*. Linux Expo. May 1999. (Updated statistics at http://immunix.org/StackGuard/performance.html)

[CWPBW00] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie and Jonathan Walpole. *Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade*. DARPA Information Survivability Conference and Exposition. January 2000.

[DLNS98] David Detlefs, K. Rustan M. Leino, Greg Nelson and James B. Saxe. *Extended Static Checking*. Research Report, Compaq Systems Research Center. December 18, 1998.

[DRS01] Nurit Dor, Michael Rodeh and Mooly Sagiv. *Cleanness Checking of String Manipulations in C Programs via Integer Analysis*. 8th International Static Analysis Symposium. To appear, July 2001.

[ES99] Úlfar Erlingsson and Fred B. Schneider. *SASI Enforcement of Security Policies: A Retrospective*. New Security Paradigms Workshop. September 1999.

[ES00] Ulfar Erlingsson and Fred B. Schneider. *IRM Enforcement of Java Stack Inspection*. IEEE Symposium on Security and Privacy. May 2000.

[ECGN99] Michael D. Ernst, Jake Cockrell, William G. Griswold and David Notkin. *Dynamically Discovering Likely Program Invariants to Support Program Evolution*. International Conference on Software Engineering. May 1999.

[EGHT94] David Evans, John Guttag, Jim Horning and Yang Meng Tan. *LCLint: A Tool for Using Specifications to Check Code.* SIGSOFT Symposium on the Foundations of Software Engineering. December 1994.

[Evans96] David Evans. *Static Detection of Dynamic Memory Errors.* SIGPLAN Conference on Programming Language Design and Implementation. May 1996.

[ET99] David Evans and Andrew Twyman. *Flexible Policy-Directed Code Safety.* IEEE Symposium on Security and Privacy. May 1999.

[Evans00a] David Evans. *Policy-Directed Code Safety.* MIT PhD Thesis. February 2000.

[Evans00b] David Evans. *Annotation-Assisted Lightweight Static Checking.* First International Workshop on Automated Program Analysis, Testing and Verification. June 2000.

[Evans00c] David Evans. *LCLint User's Guide, Version 2.5.* May 2000. http://lclint.cs.virginia.edu/guide/

[FBF99] Timothy Fraser, Lee Badger and Mark Feldman. *Hardening COTS Software with Generic Software Wrappers.* IEEE Symposium on Security and Privacy. May 1999.

[GWTB96] Ian Goldberg, David Wagner, Randi Thomas and Eric A. Brewer. *A Secure Environment for Untrusted Helper Applications: Confining the Wily Hacker.* 6th USENIX Security Symposium. July 1996.

[GH93] John V. Guttag and James J. Horning, editors, with Stephen J. Garland, Kevin D. Jones, Andrés Modet and Jennette M. Wing. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag. 1993.

[IS97] A. Ireland and J. Stark. *On the Automatic Discovery of Loop Invariants.* 4th NASA Langley Formal Methods Workshop. September 1997.

[ISO99] ISO/IEC 9899 International Standard. *Programming Languages – C.* December 1999. Approved by ANSI May 2000.

[LHSS00] David Larochelle, Yanlin Huang, Avneesh Saxena and Seejo Sebastine. *Static Detection of Buffer Overflows in C using LCLint.* Unpublished report available from the authors. May 2000.

[Luckin01] Bob Luckin. Personal communication, April 2001.

[Lundberg01] Gregory A Lundberg. Personal communication, April 2001.

[McHugh84] John McHugh. *Towards the Generation of Efficent Code form Verified Programs.* Technical Report 40, Institute for Computing Science, University of Texas at Austin PhD Thesis, 1984.

[Necula97] George C. Necula. *Proof-Carrying Code.* 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges, January 1997.

[NL96] George C. Necula and Peter Lee. *Safe Kernel Extensions Without Run-Time Checking.* 2nd Symposium on Operating Systems Design and Implementation, October 1996.

[Orcero00] David Santo Orcero. *The Code Analyzer LCLint.* Linux Journal. May 2000.

[Pethia00] Richard D. Pethia. *Bugs in Programs.* Keynote address at SIGSOFT Foundations of Software Engineering. November 2000.

[PG00] Pramode C E and Gopakumar C E. *Static Checking of C programs with LCLint.* Linux Gazette Issue 51. March 2000.

[RE89] Jon Rochlis and Mark Eichin. *With Microscope and Tweezers: the Worm from MIT's Perspective.* Communications of the ACM. June 1989.

[Snow99] Brian Snow. *Future of Security.* Panel presentation at IEEE Security and Privacy. May 1999.

[Spafford88] Eugene Spafford. *The Internet Worm Program: An Analysis.* Purdue Tech Report 832. 1988.

[SI98] J. Stark and A. Ireland. *Invariant Discovery Via Failed Proof Attempts.* 8th International Workshop on Logic Based Program Synthesis and Transformation. June 1998.

[Torvalds98] Linus Torvalds. Message archived in *Linux Weekly News.* August 1998. http://lwn.net/980806/a/linus-noexec.html

[VBKM00] John Viega, J.T. Bloch, Tadayoshi Kohno and Gary McGraw. *ITS4 : A Static Vulnerability Scanner for C and C++ Code.* Annual Computer Security Applications Conference. December 2000.

[WFBA00] David Wagner, Jeffrey S. Foster, Eric A. Brewer and Alexander Aiken. *A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities.* Network and Distributed System Security Symposium. February 2000.

[Wagner00] David Wagner. *Static Analysis and Computer Security: New Techniques for Software*

Assurance. University of California, Berkeley, PhD Thesis, 2000.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson and Susan L. Graham. *Efficient Software-Based Fault Isolation.* 14th ACM Symposium on Operating Systems Principles, 1993.

[Wegbreit75] Ben Wegbreit. *Property Extraction in Well-Founded Property Sets.* IEEE Transactions on Software Engineering, September 1975.

[WSJ01] The Wall Street Journal. *Researchers Find Software Flaw Giving Hackers Key to Web Sites.* January 30, 2001.

A. Annotated Selected C Library Functions

```
char *strcpy (char *s1, char *s2)
 /*@requires maxSet(s1) >= maxRead(s2)@*/
 /*@ensures maxRead(s1) == maxRead (s2)
      /\ result == s1@*/;

char *strncpy (char *s1, char *s2,
               size_t n)
 /*@requires maxSet(s1) >= n - 1@*/
 /*@ensures maxRead (s1) <= maxRead(s2)
      /\ maxRead (s1) <= (n - 1)
      /\ result == s1@*/;

char *strcat (char *s1, char *s2)
 /*@requires maxSet(s1)
             >= (maxRead(s1)
                 + maxRead(s2))@*/
 /*@ensures
      maxRead(s1) == maxRead(s1)
                     + maxRead(s2)
      /\ result == s1@*/;

strncat (char *s1, char *s2, int n)
/*@requires maxSet(s1)
            >= maxRead(s1) + n@*/
/*@ensures maxRead(result)
            >= maxRead(s1) + n@*/;

extern size_t strlen (char *s)
  /*@ensures result == maxRead(s)@*/;

void *calloc (size_t nobj, size_t size)
 /*@ensures maxSet(result) == nobj@*/;

void *malloc (size_t size)
 /*@ensures maxSet(result) == size@*/;
```

These annotations were determined based on ISO C standard [ISO99]. Note that the semantics of `strncpy` and `strncat` are different – `strncpy` writes exactly `n` characters to the buffer but does not guarantee that a null character is added; `strncat` appends `n` characters to the buffer and a null character. The ensures clauses reveal these differences clearly.

The full specifications for `malloc` and `calloc` also include `null` annotations on the result that indicate that they may return NULL. Existing LCLint checking detects dereferencing a potentially null pointer. As a result, the implicit actual postcondition for `malloc` is maxSet(result) == size $\vee$ result == null. LCLint does not support general disjunctions, but possibly NULL values can be handled straightforwardly.

FormatGuard: Automatic Protection From `printf` Format String Vulnerabilities

Crispin Cowan, Matt Barringer, Steve Beattie, and Greg Kroah-Hartman

WireX Communications, Inc. `http://wirex.com/`

Mike Frantzen
Purdue University

Jamie Lokier
CERN

Abstract

In June 2000, a major new class of vulnerabilities called "format bugs" was discovered when an vulnerability in WU-FTP appeared that acted *almost* like a buffer overflow, but wasn't. Since then, dozens of format string vulnerabilities have appeared. This paper describes the format bug problem, and presents FormatGuard: our proposed solution. FormatGuard is a small patch to `glibc` that provides general protection against format bugs. We show that FormatGuard is effective in protecting several real programs with format vulnerabilities against live exploits, and we show that FormatGuard imposes minimal compatibility and performance costs.

1 Introduction

In June 2000, a major new class of vulnerabilities called "format bugs" was discovered when an interesting vulnerability in WU-FTP appeared that acted *almost* like a buffer overflow, but wasn't [22]. Rather, the problem was the sudden realization that it is unsafe to allow potentially hostile input to be passed directly as the format string for calls to `printf`-like functions. The danger is that creative inclusion of % directives in the format string coupled with the lack of any effective type or argument counting in C's varargs facility allows the attacker to induce unexpected behavior in programs.

This vulnerability is made particularly dangerous by the `%n` directive, which assumes that the corresponding argument to `printf` is of type "`int *`", and writes back the number of bytes formatted so far. If the attacker crafts the format string, then they can use the `%n` directive to write an arbitrary value to an arbitrary word in the program's memory. This makes format bugs every bit as dangerous as buffer overflows [8]: the attacker can send a single packet of data to a vulnerable program, and obtain a remote (possibly `root`) shell prompt for their trouble. Since June 2000, format bugs have eclipsed buffer overflow vulnerabilities for the most common form of remote penetration vulnerability.

There are several obvious solutions to this problem, which unfortunately don't work:

Remove the `%n` feature: The `printf %n` directive is the most dangerous, because it induces `printf` to write data back to the argument list. It has been proposed that the `%n` feature simply be removed from the `printf` family of functions. Unfortunately, there exist real programs that actually use the `%n` feature (which is in the ANSI C specification [12]) so this would break an undesirable amount of software.

Permit Only Static Format Strings: Format bugs occur because the `printf` tolerates dynamic format strings. It has been proposed that `printf` be modified to insist that the format string be static. This approach fails because a large number of programs, especially those using the GNU internationalization library, generate format strings dynamically, so this too would break an undesirable amount of software.

Count the Arguments to `printf`: Because `%n` treats the corresponding argument as an `int *` an effective format bug attack must walk back up the stack to find a word that points to the right place, and/or output a sufficient number of bytes to affect the `%n` value. Thus the attacker nearly always must provide a format string that does not match the actual number of arguments presented to `printf`. If it can be done, this approach is effective in stopping format bug attacks. Unfortunately, the varargs mechanism that C employs to permit a variable number of arguments to a given function does not permit any kind of checking of either the type or count of the arguments with-

This work supported in part by DARPA contract N66001-00-C-8032.

out breaking the standard ABI for `printf`. Varargs permits the receiving functions to "pop" an arbitrary number and type of arguments off the stack, relying on the function itself to correctly interpret the contents of the stack. A "safe varargs" that passes either an argument count or an argument terminator could be built. However, this modified varargs protocol would not be compatible with any existing dynamic or static libraries and programs.

FormatGuard, our proposed solution to the format bug problem, uses a variation on argument counting. Instead of trying to do argument counting on varargs, FormatGuard uses particular properties of GNU CPP (the C PreProcessor) macro handling of variable arguments to extract the count of actual arguments. The actual count of arguments is then passed to a safe `printf` wrapper. The wrapper parses the format string to determine how many arguments to expect, and if the format string calls for more arguments than the actual number of arguments, it raises an intrusion alert and kills the process.

The rest of this paper is organized as follows. Section 2 elaborates on the `printf` format string vulnerability. Section 3 describes FormatGuard; our solution to this problem. We present security testing in Section 4, compatibility testing in Section 5, and performance testing in Section 6. Section 7 relates FormatGuard to other defenses for `printf` format string vulnerabilities. Section 8 presents our conclusions.

2 `printf` Format String Vulnerabilities

The first known discovery of format bugs was by Tymm Twillman while auditing the source code for ProFTPD 1.2.0pre6. Basic details were released to the ProFTPD maintainers and a Linux security mailing list in early September 1999, and then publicly released via BugTraq [23] later that month. Other individuals then wrote a few other format bug exploits, but they were not immediately released to the public. It wasn't until June 2000 [22] that format bugs became widely recognized, when numerous exploits for various common software packages started to surface on security mailing lists.

Format bugs occur fundamentally because C's varargs mechanism is type unsafe. Varargs provides a set of primitives for "popping" arguments off the stack. The number of bytes "popped" depends on the type of the *expected* argument. At no time is either the type or the existence of the argument checked: the function receiving the arguments is entirely responsible for popping the correct number, type, and sequence of arguments.

The `printf` family of functions (`syslog`, `printf`, `fprintf`, `sprintf`, and `snprintf`) use varargs to support the ability to output a variable number of arguments. The format string tells the function the type and sequence of arguments to pop and then format for output. The vulnerability occurs if the format string is bogus, as is the case when the format string is actually provided by the attacker.

An example of this situation occurs when a programmer writes "`printf(str)`" as a short-hand for "`printf("%s", str)`". Because this idiom is perfectly functional, and easier to type, it has been used for many years. Unfortunately, it is also vulnerable if the attacker inserts spurious % directives in the `str` string.

The `%n` directive is particularly dangerous: it assumes that the corresponding argument to `printf` is of type "`int *`", and writes back the number of bytes formatted so far into the storage pointed to by the `int *`. The result of spurious `%n` directives in `printf` format strings is that the attacker can "walk" back up the stack some number of words by inserting some number of `%d` directives, until they reach a suitable word on the stack, and treating that word as an `int *`, use a `%n` to overwrite a word *nearly anywhere* in the victim program's address space, creating substantial security problems. If buffers are of appropriate size, the attacker can also use the buffer itself as a source of words to use as the `int *` pointer, making it even easier for the attacker to use %n to modify an arbitrary word of memory.

Thus the essential features that create format vulnerabilities are the basic lack of type safety in the C programming language, the `%n` directive that induces unexpected side-effects in `printf` calls, and the casual use of un-filtered user-input as a `printf` format string due to the common assumption that this is a safe practice. Detailed descriptions of the exploitation of `printf` vulnerabilities have been written by Bouchareine [3, 4] and Newsham [14].

3 FormatGuard: Protection from Funny Format Strings

An essential part of the format string attack described in Section 2 is that the attacker provides some number of spurious `%` directives in user-input that is subsequently used as a format string for a `printf` call. FormatGuard defends against format bug attacks by comparing the number of actual arguments presented to `printf` against the number of arguments called for by the format string. If the actual number of arguments is less than the number of arguments the format string calls for, then

```
#define printf                    mikes_print(&cnt, print0

#define print0(x, args...) x ,print1(## args)
#define print1(x, args...) x+(++cnt-cnt) ,print2(## args)
#define print2(x, args...) x+(++cnt-cnt) ,print3(## args)
...
void mikes_print(int *args, char *format, ...);
```

Figure 1 Frantzen's Argument Counter

FormatGuard deems this call to be an attack, `syslog`'s the attempt, and aborts the program. As previously discussed, C's varargs mechanism does not permit argument counting, and so the trick is to find a way to count the arguments. Section 3.1 describes how FormatGuard implements argument counting with GNU CPP, and Section 3.2 describes how FormatGuard uses the argument count to implement a protected `printf` wrapper.

3.1 Counting Arguments

Frantzen first proposed the CPP method on July 25, 2000 [10]. This method exploits the way that CPP (the C PreProcessor) handles variable argument lists. Using the macro production shown in Figure 1, CPP can count the arguments by stripping the leading argument away in each production, similar to the Lisp CAR/CDDR idiom.

On September 25, 2000 Lokier [13] proposed an improved method of using CPP variable argument syntax for argument counting. Lokier's method allowed WireX to develop argument counting for FormatGuard that is recursive, reentrant, and thus thread safe, shown in Figure 2. This code function as follows:

1. The `__formatguard_counter` production serves to capture the zero-case, so that calls to `printf` containing only a null argument list are handled correctly.
2. The `__formatguard_count1` production appends a sequence of counter place holding arguments 5, 4, 3, 2, and 1. It does so by compresses the variable argument list from `__formatguard_counter` into a single token `y`.
3. Finally, `__formatguard_count2` re-expands the compressed variable argument group `y` from `__formatguard_count1`, but in doing so maps the trailing counter place holding arguments to another series of place holders, such that the first place holder from `__formatguard_count1` is mapped to the argument `n`, which in turn is the sole output of this sequence of productions.

The result of the above three productions is that place holding counter arguments are shifted to the right in proportion to the number of arguments presented to `printf` in the first place, and therefore `__formatguard_counter()` returns the count of the number of arguments presented.

The "-1" is a kludge factor to accommodate the existence of the format string itself. The `__PRETTY_FUNCTION__` macro is inserted to allow meaningful error reporting. Figure 3 presents an example, expanding an argument list of two elements: (a, b) to return a value of 2.

3.2 Protected `printf`

Figure 2 shows a definition for a `printf` macro that includes a call to the argument counter described in Section 3.1, and passes this count to a `__protected_printf` function. The purpose is to prevent the attacker from injecting spurious % directives into an un-filtered format statements, by ensuring that the number of % directives is less than or equal to the actual number of arguments provided.

Parsing `printf` format strings can be difficult. FormatGuard determines the number of % directives in a

```
#define __formatguard_counter(y...) __formatguard_count1 ( , ##y)
#define __formatguard_count1(y...) \
     __formatguard_count2 (y, 5,4,3,2,1,0)
#define __formatguard_count2(_,x0,x1,x2,x3,x4,n,ys...)  n

#define printf(x...)     \
    __protected_printf (__PRETTY_FUNCTION__, \
        __formatguard_counter(x) - 1 , ## x)
```

Figure 2 FormatGuard Implementation, Simplifed to Handle 5 or Fewer Arguments

```
formatguard_counter (a, b)
```
which gets expanded to

```
__formatguard_count1 ( , a, b)
```
which the second macro expands to

```
__formatguard_count2 (  , a, b, 5, 4, 3, 2, 1, 0)
```
The arguments to match the __formatguard_count2 rule in the following way:

```
__formatguard_count2 (  ,  a,  b , 5,  4,  3,  2,  1, 0)
                        ^  ^   ^   ^   ^   ^   ^   ^  ^
                        |  |   |   |   |   |   |   |  |
                        _  x0  x1  x2  x3  x4  n   ys...
```
Thus n gets matched to 2, which is what is returned.

Figure 3 Example Expanding the FormatGuard Macro

format string accurately (i.e. getting the same answer that `printf` will get) by borrowing the `parse_printf_format` function from the `glibc` library itself, which conveniently enough, returns exactly the number of arguments to be formatted.

If the number of % directives exceeds the number of arguments provided to `printf`, then `__protected_printf` deems a format attack to be under way. Note that the attack is *mid-way* through: the attacker has not corrupted any significant program state, but the attacker has put the victim program in an untenable position; at the very least, it is not possible to successfully complete the `printf` call. FormatGuard responds by `syslog`'ing the intrusion attempt with an entry similar to:

```
Feb  4 04:54:40 groo foo[13128]: Immu-
nixOS format error - mismatch of 2 in
printf called by main
```
where "`foo`" is the name of the victim program, "`printf`" is one of the FormatGuard-wrapped functions (`syslog`, `printf`, `fprintf`, `sprintf`, and `snprintf`) and "`main`" is the function that `printf` was called from. FormatGuard then aborts the process to prevent the attacker from taking control, similar to the way StackGuard handles buffer overflow attacks [8, 6].

3.3 FormatGuard Packaging: Modified `glibc`

In Linux-like systems, the `printf` family of functions is provided by the `glibc` library. The `__formatguard_count` macros shown in Figure 2 are inserted into the `/usr/include/stdio.h` file and the `__protected_printf` function is inserted into the `glibc` library itself. Thus FormatGuard is packaged as a modified implementation of `glibc 2.2`.

Note that, despite the packaging of FormatGuard with a library package, programs that are to benefit from FormatGuard protection must be re-compiled from source, using the FormatGuard version of `stdio.h`. In many cases, this imposes a substantial workload on people wishing to protect an entire system with FormatGuard. However, WireX has included both FormatGuard and StackGuard [8, 6] in the latest edition of Immunix Linux. Both the Immunix system and the FormatGuard implementation of `glibc` are available for download from `http://immunix.org/`

4 Security Effectiveness

FormatGuard presents several security limitations in the form of various cases that FormatGuard does not protect against, which we present in Section 4.1. Section 4.2 presents our testing of live exploits against actual vulnerabilities found in widely used software.

4.1 Security Limitations

FormatGuard fails to protect against format bugs under several circumstances. The first is if the attacker's format string undercounts or matches the actual argument count to the printf-like function, then FormatGuard will fail to detect the attack. In theory, it is possible for the attacker to employ such an attack by creatively mis-typing the arguments, e.g. treating an int argument as `double` argument. In practice, no such attacks have been constructed, and would likely be brittle.Insisting on an exact match of arguments and % directives would induce false-positives: it is quite common for code to provide more arguments than the format string specifies. There is even an example within the `glibc` code itself.

Table 1: FormatGuard Security Testing Against Live Exploits

Program	Result Without FormatGuard	Result With FormatGuard
`wu-ftpd` [22]	`root` shell	`root` shell
`cfengine` [20]	`root` shell	FormatGuard alert
`rpc.statd` [19]	`root` shell	FormatGuard alert
`LPRng` [24]	`root` shell	FormatGuard alert
`PHP 3.0.16` [17]	`httpd` shell	FormatGuard alert
`Bitchx` [26]	user shell	FormatGuard alert
`xlock` [2]	`root` shell	FormatGuard alert
`gftp`	user shell	user shell

The second limitation is that a program may take the address of printf, store it in a function pointer variable, and then call via the variable later. This sequence of events disables FormatGuard protection, because taking the address of printf does not generate an error, and the subsequent indirect call through the function pointer does not expand the macro. Fortunately, this is not a common thing to do with a printf-like function.

The third limitation is that FormatGuard cannot provide protection for programs that manually construct stacks of varargs arguments and then make direct calls to `vsprintf` (and friends). Because such programs can dynamically construct a variable list of arguments, it is not possible to count the arguments presented through static analysis.

A variation on this problem is libraries that present printf-like functions.These libraries in turn call vsprintf directly, and thus do not get FormatGuard protection. For example the `GLib` library (part of GTK+, not to be confused with `glibc`) provides a rich family of printf-like string manipulation functions. To address this class of problems, we are considering expanding FormatGuard protection beyond glibc into other libraries that provide printf-like functionality, such as GLib.

In practice, the only limitations that we have encountered are the direct calls to vsprintf and the non-glibc library calls to vsprintf, as we show in Section 4.2.

4.2 Security Testing

To test the security value of FormatGuard, we tested it against real vulnerable programs and real live exploit programs collected from the wild. The test procedure is to run the attack exploit against the vulnerable version of the program, to verify that the vulnerability is legitimate and the attack program is functional. We then recompile the vulnerable program from source, including FormatGuard protection, *without* repairing the vulnerability, and re-run the attack against the vulnerable program. Because of the level of integration effort required to deploy FormatGuard, we consider only the Immunix system, and thus consider only the vulnerabilities for the Linux/x86 platform. The results are shown in Table 1.

We note (with some irony) that wu-ftpd was the catalyst for the format string vulnerability problem [22, 5] and yet is one of the few format bugs that we found that FormatGuard does *not* stop. Investigation revealed that this is because wu-ftpd completely re-implements its own `printf` functions (as described in Section 4.1) and thus does not use the hardened `printf` functions that FormatGuard supplies. In similar fashion, FormatGuard failed to protect gftp, which uses the family `printf`-like functions found in the `GLib` library.

While this is unfortunate for wu-ftpd and for FormatGuard, it also provides interesting additional evidence that synthetic "biodiversity" in the form of n-version programming (re-implementing the same functionality by different people) does not necessarily provide resistance against common security failure modes [7]. In this case, biodiversity seems to have actually degraded security, because the semantic failure was replicated across implementations, necessitating the replication of FormatGuard protection across these implementations.

We also note (with further irony) that the PHP vulnerability [17] is only manifest in an unusual configuration that involves *extra* logging. The cause is unsafe format string handling in the call to `syslog`. The interesting factor to note is that security-conscious administrators often increase the level of logging on their systems to provide enhanced security. If, as these vulnerabilities tend to indicate, it is the case that format bugs often result from unsafe format string handling in `syslog` calls, then increasing logging levels may occasionally have the opposite from intended effect, and actually open the host to new vulnerabilities, further increasing the need for protection against format bugs.

5 Compatibility Testing

FormatGuard is intended to be highly transparent: FormatGuard protection should not cause programs to fail to compile or run, and the "false positive" rate (legitimate computation reported as format string attacks) should be asymtopic to zero. To be effective, FormatGuard needs to compile and run literally millions of lines of production C code. In this section, we describe the extent to which we have achieved these goals.

For the most part, we have succeeded. FormatGuard has been used to build the Immunix Linux distribution, which includes 500+ RPM packages, comprising millions of lines of C code. These Immunix systems have been running in production on assorted WireX servers and workstations since October 2000. These systems function normally, being not noticabley different from non-FormatGuard machines. To date, the observed false positive rate is zero. The experience has been similar to the StackGuard "eat our own dog food" experience [6].

However, FormatGuard is also less transparent than StackGuard: of the approximately 500 packages that we compiled with FormatGuard in the construction of the Immunix system, two required modification to accommodate StackGuard protection, while approximately 70 required modification to accommodate FormatGuard protection. These modifications were required to treat C programming idioms that break when CPP directives (macros and #ifdef statements) are included inside the arguments to a macro[1], as in the following C programming idiom:

```
printf("Hello world"
#ifdef X
" is X enabled"
#endif
"\n");
```

CPP expands the above code into either

```
printf("Hello world" " is X enabled"
"\n");
```

or

```
printf("Hello world" "\n");
```

which is a convenient way of conditionally compiling strings. This creates problems for FormatGuard, because FormatGuard makes `printf` a macro instead of a pure function, and CPP does not support `#ifdef` (or other CPP directives) as argument to macros, and so the above code will not work.

The work-around is to put the printf call in parentheses, which disables macro expansion, e.g. write `(printf)("Hello world")` instead of `printf("Hello world")`. This disables FormatGuard protection for this call *only*. Thus the developer must ensure that the resulting naked call to printf is safe. However, the problematic cases almost always involve static strings being conditionally compiled, so this is rarely a difficult problem.

Once code has been compiled with FormatGuard, there are additional limitations:

- Non-FormatGuard programs can link to FormatGuard libraries without problems. However, these programs to not get the benefit of FormatGuard protection, and are still vulnerable to format bugs.
- FormatGuard programs *cannot* link to non-FormatGuard libraries unless the FormatGuard version of `glibc` is present.

Thus the Immunix platform easily hosts foreign programs, but FormatGuard-protected programs do not run on foreign platforms without some intervention.

6 Performance Testing

Any run-time security defense will impose performance costs, due to additional run-time checks that it is performing. However, a security enhancement must be efficient enough that these overhead costs are minimal with respect to the defense they provide. Ideally, the cost should be below noticability for the intended user base.

FormatGuard achieves this level of performance. Overhead is only imposed on the run-time cost of calling `*printf` and `syslog` functions. Section 6.1 presents microbenchmarks that show the precise overhead imposed on calling these functions. Section 6.2 shows macrobenchmarks that measure the imposed overhead on (fairly) `printf`-intensive programs.

1. Rumor has it that the ANSI C standard [1] mandates that `printf` is not a macro. This is not true [16].

```
int main(void) {
    int i = 0;
    int counter = 100000000;

    while (i != counter) {
        printf("%s %s %s\n", "a", "b", "c");
        i++;
    }
    printf("%d\n", i); // force compiler to retain the loop
    exit(0);           // & not optimize it away
}
```

Figure 4 Microbenchmark

6.1 Microbenchmarks

We measure the marginal overhead of FormatGuard protection on `printf` calls with a tight loop as shown in Figure 4. We measured the performance of this loop in single-user mode with and without FormatGuard protection, subtract out the run time of a loop executed without the `printf` to eliminate the loop overhead, and then divide to get the %overhead. The run time with FormatGuard was 19.09 seconds, without FormatGuard was 13.97 seconds, and the loop overhead was 0.032 seconds. Thus FormatGuard imposed a marginal overhead of 37% on a trivial `printf` call.

We then repeated the above experiment, but replaced the `printf` call with one that formats a through z, rather than just three letters. The FormatGuard run time was 134.7 seconds, without FormatGuard 99 seconds, and 0.032 second loop overhead has become negligible. Thus FormatGuard imposed a marginal slowdown of 36% on a more complex `printf` call, and we conclude that FormatGuard imposes a fairly consistent 37% marginal overhead on most `printf` calls.

6.2 Macrobenchmarks

Most programs do not spend much time running the `printf` function; `printf` is an I/O function, and even programs that are I/O intensive tend to format their own data rather than using `printf`. The `printf` function is mostly used to format error-handling code. So we had some difficulty finding programs that would show measurable degradation under FormatGuard. We found such a program in man2html [25], which uses `printf` extensively to output HTML-formatted man pages.

Our test was to batch translate 79 man pages through man2html, which is 596 KB of input. The test was run multiple times in single-user mode on a system with 256 MB of RAM, so I/O overhead was minimal. The result is that the batch takes 0.685 seconds without FormatGuard, and 0.698 seconds with FormatGuard. Thus in an arguably near worst-case application scenario, FormatGuard imposes 1.3% run-time overhead. In most cases, overhead is considerably lower, often negligible.

7 Related Work

Work related to FormatGuard is divided into analysis of format string vulnerabilities, which we described in Section 2, and work to protect programs against such vulnerabilities, which we describe here.

Fundamentally, format bugs exist because of the tension between strong type checking, and convenient polymorphism. C and Pascal made opposite choices in this regard: Pascal chose the safe route of strict type checking, which means that Pascal functions can never be spoofed with this kind of attack, but also means that it is difficult to write a convenient generic I/O function like `printf` in Pascal [11]. Conversely, C chose a completely type-unsafe `varargs` mechanism that makes it impossible to statically type check a polymorphic function call.

More recent programming languages such as ML have solved this tension with *type inference*, but these techniques are difficult to apply to C programs [15, 27]. Wagner et al [21] present a compromise solution in which a "taint" *type qualifier* is added to the C language, allowing programmers to designate data as "tainted" (provided by the adversary) and the compiler tracks the data usage through the program as tainted. If tainted data is presented to printf-like functions as the format string, the compiler flags an error. The main advantage to this approach is that it detects potential vulnerabilities at compile time, rather than when the attacker tries to exploit them. The main limitation of this approach is that it is not transparent: functions that collect user-input must be manually annotated as "tainted".

Since it is problematic to properly type check C programs, more pragmatic means have emerged to deal specifically with format bugs. Alan DeKok wrote PScan [9] to scan C source code looking for potential format bugs by looking for the simple/common case of a `printf`-like function in which the *last* parameter is also the format string, *and* the format string is not static.

GCC itself has an un-documented feature where "`-Wformat=2`" will cause GCC to complain about non-static format strings. This is over-general, in that it complains about legitimate code, such as internationalization support, which uses functions to generate format strings. However, Joseph Myers has implemented an enhancement to `-Wformat` that unconditionally complains about the "`printf(foo)`" case. The functionality is essentially similar to PScan, with the advantage that it is built into the compiler, and the disadvantage that it is only available in a pre-release version of the GCC compiler.

Both PScan and the -Wformat enhancement offer the advantage that they provide static warnings, so the developer knows at compile time that there is a problem, providing an opportunity to fix the problem before the code ships. However, because these static analysis methods are heuristics, they are subject to both false negatives (missing vulnerabilities) and false positives (mis-identifying non-vulnerabilities) and thus they present an additional burden on developers. The additional burden, in turn, is problematic because developers are never actually required to use those tools, and thus may choose to omit them if they prove troublesome.

In contrast, runtime techniques present a low burden on developers (see Section 5) and uniformly improves the security assurance of applications. `libformat` [18] is a library that aborts programs if they call printf-like functions with a format string that is writable and contains a `%n` directive. This technique is often effective, but because both writable format strings and `%n` directives are legal, it can be subject to false positives.

libsafe cite{libsafe} is a library approach to defending against buffer overflow attacks. In version 2.0, libsafe has added protection against format bugs by applying their technique of the library inspecting the call stack for plausible arguments, in this instance rejecting `%n` directives that try to write to the function's return address on the stack. The strength of this approach is that, like libformat, it affords protection to binary programs, and protects against format bugs in direct calls to `vsprintf` (see Section 4.1). The limitations of libsafe are that it cannot protect code compiled with the "`no_frame_pointer`" optimization, and that it only protects against format string attacks aimed at the activation record.

FormatGuard tries to achieve some of the benefits of both static and run-time techniques. By using a source-code re-compilation technique, FormatGuard achieves high precision, resulting in few false negatives, and no false positive, presenting a very low burden on developers. Even if the original developer chose not to do anything about format vulnerabilities, an end-user of an open source product can re-compile the product with FormatGuard and gain protection from format bugs the developer failed to discover.

8 Conclusions

Format bugs are a dangerous and pervasive security problem that appeared suddenly in June 2000, and continues to be a major cause of software vulnerabilities. FormatGuard protects vulnerable programs against this problem. We have shown that FormatGuard is effective in stopping format bug attacks, imposes minimal compatibility, problems, and has a practical performance penalty of less than 2%. FormatGuard is incorporated into WireX's Immunix linux distribution and server products, and is available as a GPL'd patch to `glibc` at `http://immunix.org`

References

[1] American National Standards Institute, Inc. *Programming Language – C, ANSI Standard X3.159*. American National Standards Institute, Inc., 1989.

[2] "bind". xlock (exec) Input Validation Error. Bugtraq mailing list, `http://www.securityfocus.com/vdb/bottom.html?vid=1585`, August 15 2000.

[3] Kalou/Pascal Bouchareine. Format String Vulnerability. `http://plan9.hert.org/papers/format.html`, July 18 2000.

[4] Pascal Bouchareine. User Supplied Format String Bug. `http://julianor.tripod.com/usfs.html`, July 2000.

[5] Crispin Cowan. Format Bugs in Windows Code. Vuln-dev mailing list, `http://www.securityfocus.com/archive/82/81455`, September 10 2000.

[6] Crispin Cowan, Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle, and Erik Walthinsen. Protecting Systems from Stack Smashing Attacks with StackGuard. In *Linux Expo*, Raleigh, NC, May 1999.

[7] Crispin Cowan, Heather Hinton, Calton Pu, and Jonathan Walpole. The Cracker Patch Choice: An Analysis of Post Hoc Security Techniques. In *Proceedings of the 19th National Information Systems Security Conference (NISSC 2000)*, Baltimore, MD, October 2000.

[8] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *7th USENIX Security Conference*, pages 63–77, San Antonio, TX, January 1998.

[9] Alan DeKok. PScan: A limited problem scanner for C source files. Bugtraq mailing list, `http://www.securityfocus.com/archive/1/68688` and the web `http://www.striker.ottawa.on.ca/ aland/pscan/`, July 7 2000.

[10] Mike Frantzen. Poor Man's Solution to Format Bugs. Vuln-dev mailing list, `http://www.securityfocus.com/archive/1/72118`, July 25 2000.

[11] Brian Kernighan. Why Pascal is not my Favorite Programming Language. Report 100, AT&T Bell Labs, Murry Hill, NJ, July 1981. submitted for publication.

[12] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ, second edition, 1988.

[13] Jamie Lokier. Varargs macros subtly broken. GCC mailing list, `http://gcc.gnu.org/ml/gcc/2000-09/msg00604.html`, September 25 2000.

[14] Tim Newsham. Format String Attacks. Bugtraq mailing list, `http://www.securityfocus.com/archive/1/81565`, September 9 2000.

[15] Robert O'Callahan and Daniel Jackson. Lackwit: A Program Understanding Tool Based on Type Inference. In *Proceedings of International Conference on Software Engineering (ICSE'97)*, Boston, MA, May 1997.

[16] P.J. Plauger. *Standard C Library*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[17] "Weld Pond". @stake Advisory: PHP3/PHP4 Logging Format String Vulnerability (A 101200-1). Bugtraq mailing list, `http://www.securityfocus.com/archive/1/139259`, October 12 2000.

[18] Tim J. Robbins. libformat. `http://the.wiretapped.net/security/host-security/libformat/`, November 2001.

[19] "ron1n". statdx2 - linux rpc.statd revisited. Bugtraq mailing list, `http://marc.theaimsgroup.com/?l=bugtraq&m=97123424719960&w=2`, October 11 2000.

[20] Pekka Savola. Very probable remote root vulnerability in cfengine. Bugtraq mailing list, `http://marc.theaimsgroup.com/?l=bugtraq&m=97050677208267&w=2`, October 2 2000.

[21] Umesh Shankar, Kunal Talwar, Jeff Foster, and David Wagner. Automated Detection of Format-String Vulnerabilities. In *USENIX Security Symposium*, Washington, DC, August 2001.

[22] "tf8". Wu-Ftpd Remote Format String Stack Overwrite Vulnerability. `http://www.securityfocus.com/bid/1387`, June 22 2000.

[23] Tymm Twillman. Exploit for proftpd 1.2.0pre6. Bugtraq mailing list, `http://www.securityfocus.com/templates/archive.pike?list=1&mid=28143`, September 1999.

[24] "venomous". LPRng remote root exploit. Bugtraq mailing list, `http://marc.theaimsgroup.com/?l=bugtraq&m=97683900820267&w=2`, December 14 2000.

[25] Richard Verhoeven. man2html. `http://wsinwp01.win.tue.nl:1234/`, February 10 2000.

[26] "Zinx Verituse". BitchX - more on format bugs? Bugtraq mailing list, `http://www.securityfocus.com/archive/1/68256`, July 3 2000.

[27] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *NDSS (Network and Distributed System Security)*, San Diego, CA, February 2000.

Detecting Format String Vulnerabilities with Type Qualifiers*

Umesh Shankar Kunal Talwar Jeffrey S. Foster David Wagner
{ushankar,kunal,jfoster,daw}@cs.berkeley.edu
University of California at Berkeley

May 11, 2001

Abstract

We present a new system for automatically detecting format string security vulnerabilities in C programs using a constraint-based type-inference engine. We describe new techniques for presenting the results of such an analysis to the user in a form that makes bugs easier to find and to fix. The system has been implemented and tested on several real-world software packages. Our tests show that the system is very effective, detecting several bugs previously unknown to the authors and exhibiting a low rate of false positives in almost all cases. Many of our techniques are applicable to additional classes of security vulnerabilities, as well as other type- and constraint-based systems.

1 Introduction

Securing systems that interact with malicious parties can be a tremendous challenge. Indeed, systems written in C are especially difficult to secure, given C's tendency to sacrifice safety for efficiency. One of the more subtle pitfalls facing implementors is the so-called format string vulnerability. Since the discovery of this failure mode in the past year, security experts have identified format string vulnerabilities in dozens of widely-deployed security-critical systems [2, 4, 5, 8, 9, 10, 11, 22, 23, 24, 25, 27, 30, 35, 43], and attackers have begun exploiting these security holes on a large scale [10, 27], gaining root access on vulnerable systems. It seems likely that many legacy applications still contain undiscovered format string vulnerabilities.

Format string bugs arise from design misfeatures in the C standard library combined with a problematic implementation of variable-argument functions. Consider a typical usage of format strings:

```
printf("%s", buf);          (correct)
```

*This research was supported in part by the National Science Foundation CAREER Award No. CCR-0093337, NSF CCR-9457812, NASA Contract No. NAG2-1210, NSF CCR-0085949, a UC Regents Fellowship, and an equipment donation from Intel.

The first argument to `printf()` is a format string that specifies the number and types of the other arguments. No checking is done, either at run-time or compile-time, to verify that `printf()` was indeed called with the correct number and types of arguments. Thus the following innocuous-looking simplification of the above call can be dangerous:

```
printf(buf);          (may be incorrect!)
```

If `buf` contains a format specifier (e.g., "%s"), `printf()` will naively attempt to read nonexistent arguments off the stack, most likely causing the program to crash. The C standard library contains a number of other, similar primitives that put the programmer at risk for format string bugs. Other examples include the message-logging `syslog()` function, as well as `setproctitle()`, which sets the X window name associated with the current process.

A perhaps unexpected consequence of format string bugs is that they can be devastating to security. When a knowledgeable adversary has control of the value of the format string *s* involved in a format string bug, they can use *s* to write to arbitrary memory locations. For example, including the "%n" specifier in a format string causes `printf`-like functions to store the number of characters printed so far into a location pointed to by the associated argument. When combined with other tricks, this often leads to a complete compromise of security. Techniques for exploiting format string bugs have been described elsewhere [30]; for the purposes of this paper, the details are unimportant.

The main contribution of this paper is to describe a system for automatically detecting format string bugs at compile-time. Our system applies static, type-theoretic analysis techniques from the programming languages literature to the task of detecting potential security holes. We have implemented our system as a tool built on top of an extensible *type qualifier* framework [19]. We have tested our tool on a number of real-world software systems, in the process independently re-discovering several format string bugs that were unknown to the authors at the time.

```
while (fgets(buf, sizeof buf, f)) {
  lreply(200, buf);
  :
}

void lreply(int n, char *fmt, ...)  {
  :
  vsnprintf(buf, sizeof buf, fmt, ap);
  :
}
```

Figure 1: A format string vulnerability found in wuftpd 2.6.0, paraphrased for brevity.

Before describing the ideas behind our tool in more detail, we discuss some of the alternatives to static analysis; more are discussed in Section 6.

One natural alternative to static analysis is testing. The main weakness of testing is coverage—it is extremely difficult to construct a test suite that exercises all possible paths through a program. Unfortunately, a security auditor is most interested in exactly the paths that are never followed in ordinary operation. For example, a major source of format string bugs comes from error reporting code (e.g., calls to `syslog()`). Such code is triggered only on rare, exceptional paths, and it is easy to overlook such paths—and hence, such bugs—with run-time testing. With static analysis, on the other hand, vulnerabilities can be proactively identified and fixed before the code is ever run.

Another alternative to automated static analysis is manual code review. Unfortunately, humans are not especially good at finding format string bugs by inspection. Figure 1 shows a representative example, excerpted from a recent version of wuftpd [2, 43]. The code in Figure 1 reads a line of text from the network and passes it to `lreply()`, where it will later be used as a format string specifier to `vsnprintf()`. The correct syntax would have been `lreply(200, "%s", buf)`, but the programmer omitted the `"%s"`. As before, this introduces a serious security vulnerability.

In real code, the omission of a format string is often located far away from the place where the requirement for a trusted format string specifier becomes apparent. In the case of our wuftpd example, the offending call to `lreply()` was not even in the same file as the eventual use of `vsnprintf()`. Figure 1 also shows why naive static analysis—e.g., searching for all occurrences of `printf(s)` and replacing them with `printf("%s", s)`—does not work in practice. Very often format string bugs occur within wrapper functions to `printf()`, and these non-localized bugs require more sophisticated analysis techniques.

A third alternative would be to re-implement the application in a safe language (such as Java). However, such an approach is likely to be too costly for most legacy applications.

1.1 Type Systems for Finding Format String Bugs

Format string vulnerabilities occur when untrustworthy data (i.e., data that could potentially be controlled by an attacker) is used as a format string argument. Therefore, in our analysis we treat all program inputs that could be controlled by the adversary as "tainted," and we track the propagation of tainted data through each of the program's operations. Any variable assigned a value derived from tainted data will itself be marked as tainted, and so on. If there is any execution path in which tainted data will be interpreted as a format string by some C library function, we raise an error.

Our approach is thus conceptually similar to Perl's successful taint mode [32, 42], but with an important difference. Rather than using run-time taint propagation (which is more easily implemented for interpreted languages, such as Perl, than for compiled languages like C), we apply a static taint analysis so that we can detect bugs before the program is ever run.

We model tainting by extending the existing C type system with extra *type qualifiers*. The standard C type system already contains qualifiers such as `const`; we add a new qualifier, `tainted`, to tag data that originated from an untrustworthy source. We label the types of all untrusted inputs as tainted, e.g.,

```
tainted int getchar();
int main(int argc,
         tainted char *argv[]);
```

The first annotation specifies that the return value from `getchar()` should be considered tainted. The second specifies that the command-line arguments to the program should be treated as a tainted value.

We construct typing rules so that taint information will be propagated appropriately. Given a small set of initial tainting annotations, we infer a typing for all program variables indicating whether each variable might be assigned a value derived from a tainted source. If any expression with a `tainted` type is used as a format string, we warn the user of the potential security hole. This use of type inference for automated detection of security vulnerabilities in legacy applications is, to our knowledge, novel, and we conjecture that it may find applications elsewhere as well.

We would like to emphasize that, although in this paper we present type qualifiers in the context of finding format string bugs in C programs, in fact our implementation is expressly designed to be extensible to other kinds of type qualifiers, and indeed the idea of a type qualifier system can be applied to most standard type systems.

A key advantage to using type qualifiers is that they extend the existing type system in a backwards-compatible way. Our tool comes with default type annotations for the standard C library functions, which allows us to analyze legacy code for format string vulnerabilities with little annotation effort from the code reviewer and no modification to application source code. At the same time, type qualifiers provide a way for developers to express more detailed assertions about trust relationships in the program, and therefore programmers who are willing to spend time adding application-specific annotations can reap the extra benefits of this additional information. In other words, type qualifiers have the beneficial property that the value one obtains from the tool is proportional to the effort invested.

Type systems have several advantages over other program analysis techniques:

1. Types are a familiar way to annotate programs. We want to make it convenient for programmers to add information to their programs about tainted inputs and must-not-be-tainted variables. Type-based methods meet this goal, because programmers are accustomed to expressing invariants using types.
2. Types are a familiar way to express the output of our analysis. To be useful, when errors are reported, our tool needs to explain why the erroneous code was rejected. Giving a typing on the relevant program variables is a way to express this output in a form that programmers can readily understand.
3. Type theory is well understood. There are many efficient algorithms known in the programming languages community for inferring and manipulating types.
4. Types provide a sound basis for formal verification. Once we have found and eliminated bugs from our code, it is useful to have tools to verify that there are no format string bugs left. Because it is well-known how to build a sound type system (i.e., one where all programs that typecheck will be guaranteed free of format string bugs), types provide a single foundation that can be applied both to bug-finding and to software verification.

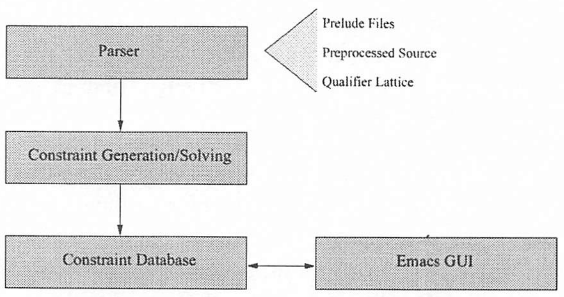


Figure 2: The architecture of the `cqual` system. The C source code and the configuration files are parsed, producing an annotated Abstract Syntax Tree (AST). `cqual` traverses the AST to generate a system (or database) of type constraints, which are solved on-line. Warnings are produced whenever an inconsistent constraint is generated. The analysis results are presented to the programmer in an emacs-based GUI, which interactively queries the constraint solver to help the user determine the cause of any error messages.

In summary, we focus our attention on type-based methods primarily because types provide a uniform, understandable interface to our tool.

Although our work relies heavily on theoretical techniques from the programming languages community, we emphasize that our efforts are aimed at providing a practical tool. Thus, we set out to build a tool that is easy to use, efficient on common hardware, effective at finding typical format string bugs, and unlikely to generate many false alarms.

2 Background

Our tool is built on top of `cqual`, a C implementation of an extensible type qualifier framework [19]. In this section we describe the underlying theory and design of `cqual`, which has broad applicability as an extension of the C type system.

2.1 System Architecture

Figure 2 shows the structure of the `cqual` tool. The main input to the tool is the preprocessed C code the user wishes to analyze. The user also provides two types of configuration files to customize `cqual` to the particular checking task. The lattice file describes the type qualifiers the user is interested in (Sections 2.2 and 2.3). The prelude files contain annotated function declarations that override the declarations in the source being analyzed.

Given preprocessed C code and configuration files, cqual performs type inference on the program (Section 2.4). Finally, the results of the type inference phase are presented to the user interactively using *Program Analysis Mode* (PAM) for emacs (Section 3).

The configuration files make cqual usable "out-of-the-box," i.e., without making any changes to the source except preprocessing. We were able to analyze all of our benchmark programs with the same standard prelude file and, in virtually all cases, no direct changes to the application source code. Typically, a few application-specific entries were added to a special local prelude file, to improve accuracy in the presence of wrappers around library functions (though the GUI indicates which ones to add). This goes a long way toward making cqual an easily usable tool.

2.2 Type Qualifiers and Subtyping

To find format string bugs, we use a type qualifier system with two qualifiers, tainted and untainted. We mark the types of values that can be controlled by an untrusted adversary with tainted. All other values are given types marked untainted. This is similar to the concept of tainting in Perl [32, 42].

Intuitively, cqual extends the type system of C to work over *qualified types*, which are the combination of some number of type qualifiers with a standard C type. We allow type qualifiers to appear on every level of a type. Examples of qualified types are int, tainted int, untainted char * (a pointer to an untainted character), and char * untainted (an untainted pointer to a character).

The key idea behind our framework is that type qualifiers naturally induce a *subtyping* relationship on qualified types. The notion of subtyping most commonly appears in object-oriented programming. In Java, for example, if B is a subclass of A (which we will write $B < A$), then an object of class B can be used wherever an object of class A is expected.

Consider the following example program:

```
(1)  void f(tainted int);
     untainted int a;
     f(a);
```

In program (1), f, which expects tainted data, is passed untainted data. In our system, this program typechecks. Intuitively, if a function can accept tainted data (presumably by doing more checks on its input), then it can certainly accept untainted data.

Now consider another program:

```
(2)  void g(untainted int);
     tainted int b;
     g(b);
```

In this case, g is declared to take an untainted int as input. Then g is called with a tainted int as a parameter. Our system should complain about this program: tainted data is being passed to a function that expects untainted data.

Putting these two examples together, we have the following subtyping relation:

$$\texttt{untainted int} < \texttt{tainted int}$$

As in object-oriented programming, if $T_1 \leq T_2$ (read T_1 is a subtype of T_2), then T_1 can be used wherever T_2 is expected, but not vice-versa. We write $T_1 < T_2$ if $T_1 \leq T_2$ and $T_1 \neq T_2$.

2.3 The Qualifier Lattice

The cqual tool needs to know not only how integer types with qualifiers relate but also how qualifiers affect pointer types, pointer-to-pointer types, function types, and so on. Fortunately, standard results on subtyping tell us how to extend the subtyping on integers to other data types [1, 28].

We supply cqual with a configuration file placing the qualifiers (in this case, tainted and untainted) in a lattice [14]. A lattice is a partial order where for each pair of elements x and y, the least upper bound and greatest lower bound of x and y both always exist. Using a lattice makes the implementation slightly easier. For finding format string bugs, we specify in the lattice configuration file that untainted < tainted.

Given this configuration file, cqual extends the supplied lattice on qualifiers to a subtyping relation on qualified C types. We have already seen one of the subtyping rules:

$$\frac{Q_1 \leq Q_2}{Q_1\ \texttt{int} \leq Q_2\ \texttt{int}}$$

This is a natural-deduction style inference rule. In general, an inference rule says that if the statements above the line are true, then the statements below the line are also true. This particular inference rule is read as follows: If $Q_1 \leq Q_2$ in the lattice (Q_1 and Q_2 are qualifiers), then Q_1 int is a subtype of Q_2 int (note the overloading of $\leq$). For our example, it means that untainted int $\leq$ tainted int. The same kind of rule applies to any primitive type (char, double, etc.).

For pointer types, we need to be a little careful. Naively, we might expect to use the following rule for pointers:

$$\frac{Q_1 \leq Q_2 \qquad T_1 \leq T_2}{Q_1\, ptr(T_1) \leq Q_2\, ptr(T_2)} \text{ (Wrong)}$$

Here the type $Q_1\, ptr(T_1)$ is a pointer to type T_1, and the pointer is qualified with Q_1. Note that τ_1 represents an extended C type, and thus may itself be decorated with tainted/untainted qualifiers. In C, the type Q_1 *ptr*(T_1) might be written

```
typedef T1 *ptr_to_t1;
typedef Q1 ptr_to_t1 q1_ptr_to_t1;
```

The rule (Wrong) says that if $Q_1 \leq Q_2$ in the lattice and T_1 is a subtype of T_2, then we can conclude that $Q_1\, ptr(T_1)$ is a subtype of $Q_2\, ptr(T_2)$.

Unfortunately, this turns out to be unsound, as illustrated by the following code fragment:

```
  tainted char *t;
  untainted char *u;

  t = u;     /* Allowed by (Wrong) */
  *t = <tainted data>;
    /* Oops! This writes tainted data
       into untainted buffer *u */
```

According to (Wrong), the first assignment `t = u` typechecks, because *ptr*(`untainted char`) is a subtype of *ptr*(`tainted char`). Then `*t` becomes an alias of `*u`, yet they have different types. Therefore we can store `tainted` data into `*u` by going through `*t`, even though `*u` is supposed to be untainted.

This is a well-known problem, and the standard solution, which is followed by `cqual`, is to use the following rule:

$$\frac{Q_1 \leq Q_2 \qquad \tau_1 = \tau_2}{Q_1\, ptr(\tau_1) \leq Q_2\, ptr(\tau_2)}$$

The key restriction here is that $\tau_1 = \tau_2$. Intuitively, this restriction says that any two objects that may be aliased must be given exactly the same type.[1] In particular, if τ_1 and τ_2 are decorated with qualifiers, the qualifiers must themselves match exactly, too.

2.4 Type Inference

So far we have concentrated on the *type checking* problem: given a program fully annotated with type specifiers on all expressions, confirm that the types are consistent. Typechecking a program is straightforward. For example, the assignment `x = y` typechecks if and only if the type of `y` is a subtype of the type of `x`. The function call `f(x)` typechecks if and only if the type of `x` is a subtype of the type of the formal parameter of `f`. More detailed rules, and a proof of soundness, can be found in [19].

The type checking system described so far, however, is not useful in practice. The problem is that it requires all types to be annotated with qualifiers: for our running example, all types would need to be marked as either `tainted` or `untainted` at every level of each type. Clearly this is an undesirable property for two reasons. First, we are interested in finding bugs in legacy code that does not have any type qualifier annotations. Second, even if we are writing a program with type qualifiers in mind, adding and maintaining annotations on every type in the program would be prohibitively expensive for programmers.

The solution to this problem is *type inference*. In this model, the user introduces a small number of annotations at key places in the program, and `cqual` infers the types of the other expressions in the program. `cqual` generates fresh *qualifier variables* (variables which range over type qualifiers) at every position in a type, constrained by any annotations specified in the program. `cqual` analyses the program and generates *subtyping constraints*—i.e., inequalities of the form $T_1 \leq T_2$ for qualified types T_1 and T_2.

A *solution* to a set of subtyping constraints is a mapping from qualifier variables to qualifiers such that all of the constraints are valid according to our subtyping rules. Thus, in our system, we solve the constraints by assigning every qualifier variable to either `tainted` or `untainted`.

In our type inference algorithm, qualifier variables are introduced at every position in a type. We write qualifier variables in italics, and name them after the corresponding program variables. The i^{th} argument of function `f` has associated qualifier variable *f_argi*, and the return value of function `f` has qualifier variable *f_ret*.

Since qualifiers are implicitly introduced on all levels of a type by the type inference algorithm, to name them we modify the name of the outermost qualifier of a type. For example, given the declaration `char *x`, `cqual` generates two qualifier variables: the variable *x* qualifies the reference `x` itself, and the variable *x_p* qualifies the location `*x`. Moreover, the programmer may also explicitly introduce named qualifier variables into the pro-

[1] Java uses the rule (Wrong) for arrays. In Java, if S is a subclass of T, then S[] is a subclass of T[], where X[] is an array of X's. Java gets away with this by inserting run-time checks at every assignment into an array to make sure the type system is not violated. Since we seek a purely static system, Java's approach is not available to us.

`tainted char *getenv(const char *name);`	$\mathit{getenv\_ret\_p} = \texttt{tainted}$
`int printf(untainted const char *fmt, ...);`	$\mathit{printf\_arg0\_p} = \texttt{untainted}$
`char *s, *t;`	
`s = getenv("LD_LIBRARY_PATH");`	$\mathit{getenv\_ret} \preceq s$
	$\mathit{getenv\_ret\_p} = \mathit{s\_p}$
`t = s;`	$s \leq t$
	$\mathit{s\_p} = \mathit{t\_p}$
`printf(t);`	$t \leq \mathit{printf\_arg0}$
	$\mathit{t\_p} \leq \mathit{printf\_arg0\_p}$

Figure 3: An example of constraint generation. The left column is a code fragment; the right column gives the inferred constraints on the qualifier variables.

gram; in this case, they begin with a dollar sign ("$") in the source code to distinguish them lexically from other tokens.

For example, after the declaration `char *x;` we assign the qualified type $\mathit{x\_p}$ `char *` x to x. Similarly, a function declared with the prototype

```
tainted char *getenv(char *name);
```

is assigned the following fully qualified type:

$$\mathit{getenv\_ret\_p}\ \texttt{char *}\ \mathit{getenv\_ret}$$
$$\texttt{getenv}(\mathit{getenv\_arg0\_p}\ \texttt{char *}\ \mathit{getenv\_arg0}\ \texttt{name});$$
$$\text{(where } \mathit{getenv\_ret\_p} = \texttt{tainted})$$

If we then encounter an assignment `x = getenv(...)`, our type inference algorithm will conclude that the type of `getenv()`'s return value must be a subtype of the type of x, i.e.,

$$\mathit{getenv\_ret\_p}\ \texttt{char *}\ \mathit{getenv\_ret} \preceq \mathit{x\_p}\ \texttt{char *}\ x\ .$$

As a consequence, we can infer (using the subtyping rules introduced in Section 2.2 and 2.3) that we must have the following constraints on the qualifier variables:

$$\mathit{getenv\_ret\_p} = \mathit{x\_p} = \texttt{tainted}, \quad \mathit{getenv\_ret} \leq x.$$

In essence, our declaration of `getenv()` has ensured that whatever it returns will be labeled as tainted. Note that this might be used to model, for instance, a scenario where environment variables are under the adversary's control.

We give next a more detailed example. Figure 3 shows a fragment of code that manipulates tainted data in an unsafe way, along with the typing constraints generated by the type inference algorithm. The constraint $\mathit{getenv\_ret\_p} = \mathit{s\_p}$ encodes the conclusion that the return value of `getenv()` is treated as tainted (as discussed above). The prototype for `printf()` (typically found in the global prelude file) specifies that `printf()` must not be called with a tainted format string argument, by requiring that its first argument be a subtype of `untainted char *`.

The call `s = getenv("LD_LIBRARY_PATH")` generates the constraints

$$\mathit{getenv\_ret} \leq s$$
$$\mathit{getenv\_ret\_p} = \mathit{s\_p}$$

Notice the equality constraint, arising from our corrected rule for subtyping pointer types. The assignment `t = s` generates a similar constraint. Finally, the call `printf(t)` generates a subtyping constraint on the $\mathit{printf\_arg0\_p}$ because `printf`'s first argument is `const` (see Section 4.4).

Taking the transitive closure of these constraints, we have a chain of deductions

$$\texttt{tainted} = \mathit{getenv\_ret\_p} = \mathit{s\_p} = \mathit{t\_p} \leq \mathit{printf\_arg0\_p} = \texttt{untainted},$$

implying that for this example to type check, we would need `tainted` $\leq$ `untainted`. As explained in Section 2.2, this does not hold in our lattice, so this code fragment does not type check, indicating a possible format string bug. This demonstrates how our type inference algorithm can be used to identify unsafe manipulation of format strings.

In our implementation, the subtyping constraints are solved on-line as they are generated. If the constraint system ever becomes unsatisfiable, an error is flagged at the first illegal expression in the code. This allows us to pinpoint the location of unsafe operations on tainted data. The inference then continues after any errors, though in this case the quality of the remaining error messages can vary tremendously.

We observe that efficient algorithms for this type inference problem are known. Given a fixed-size qualifier lattice and n constraints of the form $l \leq q$, $q \leq l$, or

```
tainted char *getenv(const char *name);
int printf(untainted char *fmt, ...);

/* Point 1 */
char *f3(char *s) { return s; }

/* Point 2 */
char *f2(char *s) { return f3(s); }

/* Point 3 */
char *f1(char *s) { return f2(s); }

int main()
{
    char *s, *unclean;

    /* Point 0 */
    unclean = getenv("PATH");

    s = f1(unclean);   /* Point 4 */
    printf(s);         /* Point 5 */
}
```

Figure 4: An example of a taint flow path. The string `unclean` is tainted by the call to `getenv` at Point 0, and ultimately that data is passed to `printf` at Point 5.

$q_1 \leq q_2$, where l is a lattice element and q, q_1, and q_2 are qualifier variables, a solution to the constraints can be computed in $O(n)$ time using well-known algorithms [21]. The idea is to express these constraints as a directed graph with qualifier variables as vertices and subtyping constraints as directed edges: the constraint $v_1 \leq v_2$ induces an edge from v_1 to v_2. The constant qualifiers `tainted` and `untainted` are also vertices in this graph, and a directed path from `tainted` to `untainted` corresponds to a possible format string bug. We call this path a *taint flow path*. See Figure 4 for an example.

3 User Interface

Thus far, we have presented the theory underlying our tool. For a program analysis to be useful, however, one needs both a sound theoretical foundation and an intuitive, efficient interface for understanding the results.

In the folklore of type inference, it is well known that the more powerful a type inference system is, the harder it is to understand why a program contains a type error. For example, type errors from a C compiler, which performs little inference, are easy to localize. The compiler simply reports the line number where the type error occurred, and this is almost always enough to tell the programmer why the error occurred.

In our type qualifier system, however, type errors occur at the point where the type constraint system becomes unsatisfiable, and that point can be distant from the actual source of the problem. Again, consider Figure 4. In this example, the string `unclean` is tainted by the call to `getenv` at Point 0, and ultimately that data is passed to `printf` at Point 5. Given this input program, our system will warn the user of a potential format string bug at point 5. But program points 1–5 are all involved in the error, and to understand and fix the error a programmer may need to examine all five program points. In general these program points could be spread across multiple files.

Thus reporting line numbers with error messages is no longer enough. In this section, we describe the techniques we use to display the results of our tainting analysis to the user. We emphasize that without the GUI described in this section, performing the experiments described in Section 5 would have been extremely difficult.

3.1 Program Analysis Mode

Our tool `cqual` presents the results of the tainting analysis to the programmer using *Program Analysis Mode* (PAM) for Emacs [20], a GUI developed at Berkeley that is designed to add hyperlinks and color mark-ups to the preprocessed text of the program.

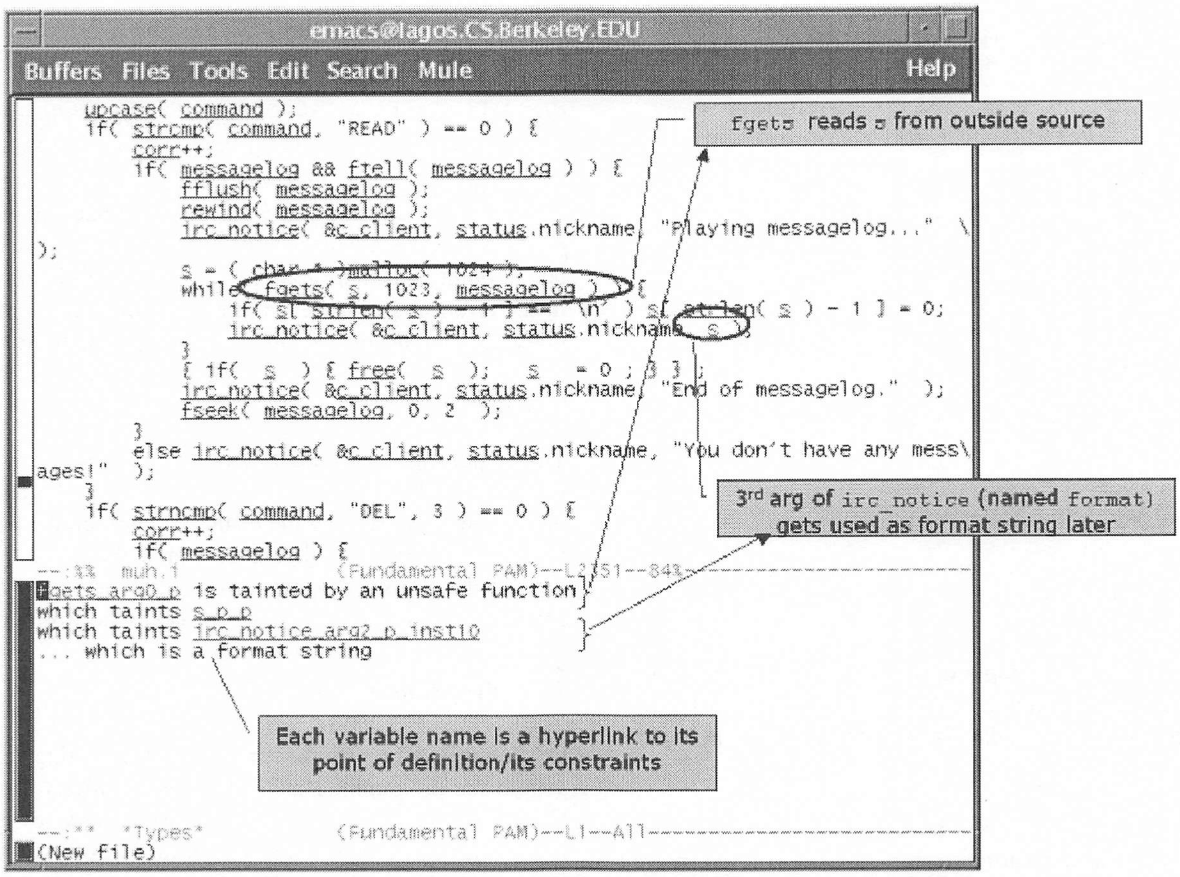
Figure 5 shows a screenshot of a run of `cqual` on muh, an IRC proxy application. `cqual` initially displays a list of all files analyzed and any errors that occurred. The user can click on a filename to jump to that file or click on an error message to jump to information about that error (see below).

Each identifier in a file is colored according to its inferred qualifiers. Tainted identifiers (those whose type contains a tainted qualifier somewhere) are colored red, untainted identifiers are colored green, and any identifiers that could be either tainted or untainted are not colored. Intuitively, this last set of qualifiers could all be marked untainted, but it is easier on the user to reduce the number of marked up identifiers.

The user can click on an identifier to display its fully qualified type, with each individual qualifier colored according to its taintedness.

3.2 Added Features

Beyond the basic coloring of qualifiers, we designed several extensions to make it easy to find and fix potential format string bugs. Many of these features are applicable to other kinds of qualifiers, and perhaps to


emacs@lagos.CS.Berkeley.EDU
Buffers Files Tools Edit Search Mule Help
fgets reads s from outside source
3rd arg of irc_notice (named format) gets used as format string later
Each variable name is a hyperlink to its point of definition/its constraints

```
    upcase( command );
    if( strcmp( command, "READ" ) == 0 ) {
        corr++;
        if( messagelog && ftell( messagelog ) ) {
            fflush( messagelog );
            rewind( messagelog );
            irc_notice( &c_client, status.nickname, "Playing messagelog..."  \
);
            s = ( char * )malloc( 1024 );
            while( fgets( s, 1023, messagelog ) ) {
                if( s[ strlen( s ) - 1 ] == '\n' ) s[ strlen( s ) - 1 ] = 0;
                irc_notice( &c_client, status.nickname, s );
            }
            { if(  s  ) { free(  s  );   s   = 0 ; } } ;
            irc_notice( &c_client, status.nickname, "End of messagelog."  );
            fseek( messagelog, 0, 2  );
        }
        else irc_notice( &c_client, status.nickname, "You don't have any mess\
ages!"  );
    }
    if( strncmp( command, "DEL", 3 ) == 0 ) {
        corr++;
        if( messagelog ) {
--:%%  muh.1           (Fundamental PAM)--L2751--84%-----------------------
fgets_arg0_p is tainted by an unsafe function
which taints s_p_p
which taints irc_notice_arg2_p_inst10
... which is a format string

--:**  *Types*         (Fundamental PAM)--L1--All---------------------------
(New file)
```

Figure 5: Screenshot of a run of cqual on the muh application.

other kinds of type inference systems as well.

Taint Flow Paths. Recall from Section 2.4 that the subtyping constraints can be thought of as inducing a directed graph among qualifiers. A path in the constraint graph from tainted to untainted indicates a type error.

For each type error, we provide a hyperlink to a display of the particular path from tainted to untainted that caused that error. Since each path in the constraint graph typically corresponds to a flow of data through the program, this helps identify the unsafe sequence of operations that lead to a type error. However, since there are typically many such paths (and possibly even cycles) in the constraint graph, displaying all of them may overload the user. Therefore, to reduce the burden on the user, we display the shortest such path, as computed with a breadth-first search. In our experience, this heuristic is very important for usability.

Figure 5 shows one example. Each qualifier in the path is hyperlinked to the definition of the identifier with that qualifier, which makes it easy to navigate the source code to determine the cause of the error.

Unannotated Functions. Our standard prelude files contain annotated versions of most standard library functions. Programs, of course, can also use system- and application-dependent libraries. In order to have a sound inference, the user must provide annotated declarations of these libraries.

To make it easy for the user to find and annotate these functions, we generate a list of hyperlinks to declarations of functions that have neither been defined nor have been declared in a prelude file.

A common idiom in many programs is to write functions that simply massage their inputs and then call a library function. For example, a program might contain a function log_error(fmt, ...) that calls fprintf(stderr, fmt, ...). As described in Section 4.3, for soundness and to improve the precision of the analysis the user should add annotations to such wrapper functions around potentially-vulnerable library calls. To aid in the annotation process we provide a hyperlinked list of unannotated variable argument functions to the user.

Hotspots. Although many of the features of the system are geared toward reducing false positives and, where there are real bugs, reducing the number of resulting warnings, occasionally the user will be faced with hundreds of warnings.

To help the user decide which warnings to investigate first, we attempt to determine "hotspots" in the code. For each error message, we compute the shortest taint flow path and increment a counter associated with each qualifier on the path. We then present the user with a hyperlinked list of the "hottest" qualifiers, i.e., those involved in the largest number of (shortest) taint flow paths. The idea—borne out by our experience—is that adding a single annotation at an important point can dramatically reduce the number of warnings.

One extension to this idea, which we have not yet implemented, is to find the hottest constraints rather than the hottest qualifiers. This may help point the user to a particular erroneous expression in the code, rather than to an identifier.

4 Finding Format String Bugs

In Section 2 we described the basic workings of the `cqual` tool. In this section we discuss extensions to make the basic tool sound in the presence of type casts and variable argument functions, and to decrease false positives by using the programmer's knowledge about the program being analyzed.

4.1 Leaf Polymorphism

Type inference is a powerful tool for computing qualifiers given a few annotations. However, when inferring types of functions, we need to introduce some new machinery to avoid getting a large number of false positives.

To understand the problem, consider the following simple example code:

```
char id(char x) { return x; }
...
tainted char t;
untainted char u;
char a, b;
a = id(t); /* 1 */
b = id(u); /* 2 */
```

Because of call 1, we infer that `x` is a `tainted char`, and therefore we also infer that `a` is `tainted`. Then call 2 typechecks (because `untainted char` $\leq$ `tainted char`), but we infer that `b` must also be `tainted`.

While this is a sound inference, it is clearly overly conservative. Even though this simple example looks unrealistic, we encounter this problem in practice, most notably with library functions such as `strcpy`. This leads to a large number of false positives.

The problem arises because we are summarizing multiple stack frames for distinct calls to `id` with a single function type—`x` has to either be untainted everywhere or tainted everywhere. The solution to this problem is to introduce *polymorphism*, which is a form of context-sensitivity.

A function is said to be *polymorphic* if it has more than one type. Notice that `id` behaves the same way no matter what qualifier is on its argument `x`: it always returns exactly `x`. Thus we can give `id` the signature α `char id(`α `char x)` for any qualifier α.

Operationally, when we call a polymorphic function, we *instantiate* its type—we make a copy of its type, replacing all the generic qualifier variables α with fresh qualifier variables. Intuitively, this corresponds exactly to inlining the function, except that instead of making a fresh copy of the function's code, we make a fresh copy of the function's type.

We need a way to write down polymorphic type signatures—for example, we should be able to express that if the `strcat()` function is passed a `tainted` second argument, then its first argument should also be `tainted`, but not vice versa. We can do this by writing a polymorphic type with side constraints on the qualifiers:

```
α char *
strcat(α char *dst, β const char *src);
                    (where α ≥ β)
```

More generally, we want to be able to specify a polymorphic function

```
α f(β arg0, δ arg1, ... );
```

with some arbitrary inequality constraints on the qualifier variables α, β, δ, etc. We define a concise notation for expressing these inequality constraints by using the following theorem.

Theorem 4.1 *Let $(P, \leq)$ be any finite partial order. Let $(2^{\mathbb{N}}, \subseteq)$ be the lattice of subsets of $\mathbb{N}$ with the set inclusion ordering. Then $(P, \leq)$ can be embedded in $(2^{\mathbb{N}}, \subseteq)$, i.e., there exists a mapping $\phi : P \rightarrow 2^{\mathbb{N}}$, such that $a \leq b \iff \phi(a) \subseteq \phi(b)$ and $\phi(x)$ is a finite subset of $\mathbb{N}$ for all $x \in P$.*

The theorem is formally proved in the appendix, and may be viewed as a concrete example of the Dedekind-MacNeille Completion [14].

This theorem enables us to define the partial order implicitly by the naming of the qualifier variables on the function arguments and return type. We represent a qualifier a in the partial order P by $\phi(a)$, which we denote as a '_' separated string of the integers in the set. If $\phi(a) = \{1, 2\}$, then a is represented as $_1_2. Hence, the polymorphic declaration of `strcat` can now be written as

```
$_1_2 char *
strcat($_1_2 char *, $_1 const char *)
```

which means that the qualifier on the return type is the same as the qualifier on the first argument, and that they are both supertypes of the second argument. In other words, since $\{1, 2\} \supseteq \{1\}$, the names of the qualifiers encode the implicit inequality constraint $_1_2 ≥ $_1. Hence for any instantiation of `strcat()`, we have

$$\begin{aligned} \texttt{strcat\_ret\_p} &= \texttt{strcat\_arg0\_p} \\ &\geq \texttt{strcat\_arg1\_p}. \end{aligned}$$

This avoids the need to write subtyping constraints on the side for each polymorphic function. Instead, the constraints are encoded implicitly in the annotations themselves, which provides a concise framework for expressing subtyping annotations.

To keep our implementation simple, we only support polymorphism for library functions, i.e., functions with no code. To be more precise, any function may be declared polymorphically, but the polymorphic prototype will not be typechecked against its implementation. This restriction is not fundamental; there are well-known efficient algorithms for more general polymorphism [19, 33]. Our standard prelude files contain appropriate polymorphic declarations for most of the standard library functions.

4.2 Explicit Type Casts

The treatment of type casts in the program's source code is very important to the correct operation of our tool. In most cases, a pointer cast is used to implement generic functions, to emulate object subtyping, or to otherwise bypass the limitations of the C type system. Since a pointer cast usually preserves the semantic meaning of the data being pointed to, we want to preserve the taintedness of data through ordinary C typecasts. Consider the following program fragment:

```
void *y;
char *x = (char *) y;
```

If `y` is tainted, then `x` should also be tainted, even though their types do not otherwise match.

Casts to `void *` are particularly problematic because one can cast any type to a `void *`. For example, a programmer might write

```
char **s, **t;
void *v = (void *) s;
t = (char **) v;
```

Here the type structure of `v` has two levels, while the type structure of `s` has three. Hence there is no direct correspondence between the qualifiers of the two types.

We solve this problem by "collapsing" qualifiers at a type cast. If we cast a type t to a type u, then we match up the qualifiers level-by-level between t and u as deeply as possible. For example, when casting `char *x` to a `void *`, we add the constraints $x \leq cast$ and $x\_p = cast\_p$, where *cast* is the name we use for the qualifiers on the `void *`. As soon the structures of types t and u diverge, we equate all the remaining qualifiers. For example, when casting a `char **x` to a `void *`, we add the constraints $x \leq cast$ and $x\_p = x\_p\_p = cast\_p$. Putting this together, in the above example if if either `*s` or `**s` is tainted, then `*v` becomes tainted. When `v` is cast to `char **t`, both `*t` and `**t` will become tainted.

We also allow the knowledgeable programmer to indicate that some program data has been validated and should consequently be considered untainted despite its origins. Such an annotation can be expressed in our system by writing an explicit cast to an `untainted` type. To enable this, we do not add any constraints in case of an explicit cast containing a qualifier. For example, in the following code

```
void *y;
char *x = (untainted char *) y;
```

the assignment does not taint `x`, regardless of the inferred taintedness of `y`.

This feature allows the security-aware developer to implement functions that parse an input string and filter out dangerous substrings without departing from our framework. We assume that the programmer will add such an annotation only after ensuring that the string is validated by some rigorous checking procedure. There is no way to verify this assumption automatically. However, our syntax is designed to make it easy to manually audit all such annotations, since they can typically be easily identified by simply `grep`ing the source code for the keyword `untainted`.

Collapsing the qualifiers at casts is conservative but sound for the most common casts in a program. There are two ways in which our implementation is currently unsound with respect to casts. First, we have found that if we collapse qualifiers on structure fields at type casts, the analysis generates too many false positives (too much becomes tainted). Thus in our implementation if one aggregate is cast to another, we ignore the cast and do not collapse type qualifiers.

Second, because we use a subtyping-based system, the qualifier-collapsing trick does not fully model casts from pointers to integers. Consider the following code:

```
char *x, *y;
int a, b;

a = (int) x;        (1)
b = a;              (2)
y = (char *) b;     (3)
```

For line (1), we generate the constraints $x\_p = x = a$. For line (2), we generate the constraint $a \leq b$. And for line (3), we generate the constraints $b = y\_p = y$. Notice that we have $x\_p \leq y\_p$ but we do not have $y\_p \leq x\_p$, so our deductions are unsound.

We leave as future work the solution to these problems. We believe that the best solution will be to combine techniques that attempt to recover the semantic behavior of casts with conservative alias analysis for ill-behaved casts [12, 36, 37].

4.3 Variable Argument Functions

C allows functions to have a variable number of arguments, through the *varargs* language feature. However, there is no obvious way of specifying constraints on the individual varargs: even their type is not fixed. For example, in the expression `sprintf(s, "%s", t)`, if `t` is tainted, then we would like our type inference algorithm to force `s` to be tainted as well.

We have extended the C grammar so that the varargs specifier "`...`" can be annotated with a type qualifier variable. In the `sprintf()` example, we would like the first argument of `sprintf()` to be tainted if any of its varargs is tainted, so we use the type declaration

```
int sprintf($_1_2 char *,
        untainted char *, $_2 ...);
```

Consequently, if any of `sprintf()`'s arguments (excluding the first two) are tainted, we will infer that the first argument must be tainted as well. More precisely, for each qualifier q on any level of a type passed to the `...` of `sprintf()`, we add the constraint $q \leq$ `$_1_2`.

The type inference system ignores parameters beyond the last named argument of an unannotated varargs function. Thus for soundness the user must annotate all potentially-vulnerable varargs functions; as mentioned in Section 3.2, we provide a list of unannotated varargs functions to the user to help with this task. Our implementation also does not model varargs function pointers fully. Both of these issues can be easily addressed, and we plan to do so in the future.

4.4 `const` Allows Deep Subtyping

As described in Section 2.3, we use a conservative rule for pointer subtyping. This rule can lead to non-intuitive *reverse taint flow*, which often causes false positives. For example, consider the following code:

```
f(const char *x);
char *unclean, *clean;
unclean = getenv("PATH");
f(unclean);
f(clean); /* 'clean' gets tainted */
```

Here the `getenv()` function call imposes the condition $unclean\_p =$ `tainted`. The first call to `f` adds the constraint $f\_arg0\_p = unclean\_p$. The second function call generates the constraint $f\_arg0\_p = clean\_p$, thereby marking `*clean` as tainted, which is counter-intuitive.

Observe, however, that `f`'s argument `x` is of type `const char *`, so `f` can not make `*x` tainted if it is not tainted in the first place. Consequently, we modify the constraints in Section 2 as follows: For an assignment

```
const char *s;
char *t;
...
s = t;
```

we add the constraints $t \leq s$ and $t\_p \leq s\_p$, if `*s` has a `const` qualifier. This is to be compared with the constraint $s\_p = t\_p$ which we would otherwise have imposed. In this way we can use "deep subtyping" to improve precision for formal parameters marked `const`.

This extra precision, which helps avoid many false positives (especially in library functions), is the main reason we work in a subtyping system. Note that we rely on the C compiler to warn the programmer about any casts which discard the `const` qualifier, i.e., we assume that a variable that is `const` is never cast to anything that is not `const`.

5 Real-World Tests

We tested the effectiveness of `cqual` on several popular C programs that are potentially vulnerable to format string attacks. Some of them had known vulnerabilities; others did not. In all cases, attackers from across the network have control over some string input to the program. If this input is used as a format string, a carefully chosen input can crash the program or give the attacker root access.

5.1 Metrics

The ideal bug detector would detect all extant bugs without flagging correct code as being incorrect. The initial output from `cqual` is a list of warnings that indicate a type error somewhere in the program. Some of these correspond to real bugs; others are *false positives* stemming from our conservative tainting approach (and lack of full polymorphism). *False negatives* are also of interest: we would like all vulnerabilities to show up as warnings. One complicating factor is that many warnings can result from the same bug—for example, if many functions reading network data call a single function that has a format string bug, then all the warnings may go away when that bug is fixed.

We chose the following metrics, measured per-program:

- How many known vulnerabilities were detected and how many went undetected?
- How many false positives were there?
- How easy was it to check whether a warning was a real bug?
- How long did the automatic analysis take, and what were its resource needs?
- How easy was it to prepare programs for analysis?

5.2 Test Setup

Testing was performed on a dual-processor 550MHz Pentium III Xeon machine running the Linux 2.2.16-3smp kernel. Only one processor was used in testing. The machine had 2GB of memory. Tools used in preparation and testing were gcc, version egcs-2.91.66; emacs, version 20.7.1; and PAM (Program Analysis Mode for emacs), version 3. Some programs were prepared (preprocessed) on an UltraSparc-based machine running Solaris 7 and gcc 2.95.2.

To test our system, we chose several widely-used daemons written in C that were likely to contain security vulnerabilities. We also included several programs with reported format string bugs in order to test the coverage (false negative rate) of our system. Two of these cases—mingetty [24] and mars_nwe [25]—are particularly interesting because hand audits had revealed potentially dangerous function calls, but owing to the difficulty of manual verification, no actual bugs had been reported. In some other cases, such as cfengine [35] and bftpd [4], we detected bugs that were unknown to us at the time of the experiment, but that we later discovered had already been known to others.

5.3 Results

Following is a brief description of the analysis results on some test samples:

cfengine: The first run gave many warnings; hotspot analysis led to a real format string vulnerability previously unknown to us. The vulnerability turned out to be known to others [35]. In addition, there were a few warnings unrelated to taint analysis.

muh: The first run generated many warnings. After looking at the hotspots and the list of unannotated functions, six library function wrappers were annotated with polymorphic types in the local prelude file. A subsequent run showed twelve warnings, one of which was a real vulnerability (known to others [22]).

bftpd: The hotspots from the first run guided us to mark one function with a polymorphic type. After this, there were two warnings, one of which was a bug of which we were not previously aware. We later found that this bug had already been discovered by others [4].

mars_nwe: In the first run, there were a few hundred warnings, but the hotspots suggested making two functions polymorphic. When this was done, there were no more warnings. Note that others had previously reported questionable function calls where the auditor was not able to determine whether the property could be exploited [25]; our tool gives strong evidence that they are not exploitable.

Name	Version	Description	Lines	Preproc.	Time	Warnings	Bugs
cfengine	1.5.4	System administration tool	24k	126k	28s	5	1
muh	2.05d	IRC proxy	3k	103k	5s	12	1
bftpd	1.0.11	FTP server	2k	34k	2s	2	1
mars_nwe	0.99	Novell Netware emulator	21k	73k	21s	0	0
mingetty	0.9.4	Remote terminal control utility	0.2k	2k	1s	0	0
apache	1.3.12	HTTP server	33k	136k	43s	0	0
sshd	2.3.0p1	OpenSSH ssh daemon	26k	221k	115s	0	0
imapd	4.7c	Univ. of Wash. IMAP4 server	43k	82k	268s	0	0
ipopd	4.7c	Univ. of Wash. POP3 server	40k	78k	373s	0	0
identd	1.0.0	Network identification service	0.2k	1.2k	3s	0	0

Figure 6: Results of our experimental evaluation of the tool. The size of the program is measured unpreprocessed and preprocessed, in thousands of lines of code, excluding comments. Time is the wall clock time for a run of `cqual`. Warnings counts the total number of warnings issued by `cqual` after the GUI's recommendations were followed, and Bugs is the number of real vulnerabilities found.

mingetty: No warnings issued. As with mars_nwe, an auditor had previously reported a suspicious function call of unknown exploitability [24]; `cqual` made it easy to verify that these calls were safe.

apache: In the first two runs, there were some warnings due to inconsistent declarations in the prelude and the source files. After these were set right, no warnings were issued.

sshd: The first run suggested annotation of twelve vararg functions. After these were made polymorphic, there were no more warnings.

imapd, ipopd, and **identd:** No warnings issued.

5.4 Evaluation

Our system reliably found all known bugs in the tested programs, including bugs we were not aware of when we applied our tool. Code without known bugs, and which was later examined by hand and found to be unlikely to contain bugs, yielded few false positives. Indeed, in our tests all false positives occurred in programs with actual bugs once varargs functions were annotated. The heuristics described in Section 3.2 were extremely useful in such cases. The annotation of varargs functions flagged by `cqual` was usually enough to remove most false positives. The hotspots pinpointed the actual bug in most cases. The GUI was invaluable in the analysis, making quick detection and correction of bugs possible. The source of most bugs was found within a few minutes of manual inspection of unfamiliar code. Thus, our experience shows that false positives—a common drawback of many tools based on static analysis—do not seem to be a problem in our application.

The automated analysis usually took less than a minute, and never more than ten minutes. The manual effort required for each program was usually within a few tens of minutes.

Preparation of the programs for analysis typically took between thirty and sixty minutes each. Note that we were not familiar with the layout and particular structure of the source code for any of the test programs. Preparation consisted of modifying the build process to output preprocessed, filtered source. In practice this could be more systematically added to the build process.

In summary, we evaluated our tool on a number of security-sensitive applications, demonstrating the ability of our tool to find security holes that we were not previously aware of. We feel that this validates the power of our approach.

6 Related Work

Lexical Techniques. `pscan` [15] is a simple tool for automatically scanning source code for format string vulnerabilities. `pscan` searches the input source code for lexical occurrences of function calls syntactically similar to, e.g., `sprintf(buffer, variable)`. Because `pscan` operates only on the lexical level, it cannot reason about the flow of values through the program and fails in the presence of wrappers around C libraries (see, e.g., Figure 1). `pscan` also cannot distinguish between safe calls when the format string is a variable and unsafe calls—it flags any call where a format string is non-constant.

Others have exploited lexical source code analysis to find security bugs [7, 38]. The main advantages of lexical analysis are that it is extremely fast, it can find bugs in non-preprocessed source files, and it is virtually language independent. However, because lexical tools have no knowledge of language semantics, many errors—such as those involving aliasing or non-local control paths—cannot be detected.

Taint Analysis. Our use of tainting, inspired by Perl's taint mode [32], bears some resemblance to a Biba integrity model [6] and thus is distantly related to previous work on enforcing information flow policies through typing [29, 39, 40]. However, because we do not have to deal with maliciously constructed code, we avoid the need to solve many of the most vexing challenges (e.g., covert channels) in enforcing information flow policies.

Type Qualifiers. The basic framework for type qualifiers, as presented in Section 2, is due to Foster et al. [19] and has been used to build Carillon, a tool for finding Y2K bugs in C programs [16]. As described in Section 4, we developed several refinements to make tainting analysis practical: improved handling of casts and variable-argument functions; the notation for polymorphic type signatures; and the improved user-interface. However, the one feature present in previous tools that is missing from our system is automated type inference of polymorphic types for all functions. We are planning to incorporate polymorphic recursion [33] in the future to remedy this.

Static Bug Detection. Many authors have noted that static analysis can be a useful tool for detecting bugs. For instance, LCLint [18] uses dataflow analysis to search for common errors in C programs; Engler et al.'s Meta-level Compilation [17] statically simulates the behavior of a user-defined finite state machine and has been successful at finding many new bugs; and the Extended Static Checking system (ESC) [26] uses theorem proving to verify the validity of annotated Java source code.

These systems have been very successful at detecting many common bugs. However, they are not well suited to detecting format string vulnerabilities, for two reasons. First, they focus primarily on local properties, whereas format string vulnerabilities often arise due to global mishandling of strings. Second, many of them (e.g., ESC and, to a lesser degree, LCLint) require extensive annotations from the user, which we would like to avoid. Our type-based techniques address these challenges directly.

Run-time Techniques. Another defense against format string vulnerabilities is to dynamically prevent exploits through appropriate modifications to the C runtime [3], compiler, or libraries. `libformat`, a library designed to halt execution of any program that might be susceptible to a format string bug, follows this approach: it intercepts calls to `printf`-like functions and aborts the application if the format string specifier contains `%n` and the format string is in a writable portion of the address space [34]. However, this approach is fragile, since the `libformat` mechanism must be kept in perfect synchronization with the `libc` implementation of all `printf`-like functions.

FormatGuard, a compiler modification, injects code to dynamically check and reject all `printf`-like function calls where the number of arguments does not match the number of "`%`" specifiers [13]. Of course, only applications that are re-compiled using FormatGuard will benefit from its protection. Also, one technical shortcoming of FormatGuard is that it does not protect user-defined wrapper functions (see, e.g., Figure 1).

Moreover, a common limitation of both `libformat` and FormatGuard is that programs with format string vulnerabilities remain vulnerable to denial of service attacks. Nonetheless, an important advantage of these runtime techniques is that they are cheap and require almost no human intervention. Thus, we feel that run-time and static measures are both useful and complement each other well.

7 Conclusions

We have described a tool for automated detection of format string vulnerabilities in legacy source code. We have shown that our tool has very low false positive and false negative rates and is useful in practice at detecting even security holes that were unknown to us. Therefore, we feel that our work represents a strong step toward a usable bug-detection system.

The key technique we exploit is type qualifier inference, applied to the problem of static taint analysis. This approach allowed us to scale to large programs with hundreds of thousands of lines of code and to present an intuitive user interface to the programmer. Consequently, we conjecture that these techniques may find use in future applications as well.

Acknowledgments

We would like to thank David Gay for developing the C front end we used, Chris Harrelson and Henning Niss for developing PAM, Martin Elsman for his work on Carillon, the precursor to `cqual`, and the anonymous referees for their comments.

References

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.

[2] Lamagra Argamal. "ftpd: the advisory version." bugtraq mailing list, 23 June 2000. http://www.securityfocus.com/archive/1/66544.

[3] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. "Efficient Detection of All Pointer and Array Access Errors." In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, June 1994.

[4] Christophe Bailleux. "Asynchro," bugtraq mailing list, 8 December 2000. http://www.securityfocus.com/archive/1/149977.

[5] D.J. Bernstein, "Re: Logging question." qmail mailing list, 13 September 1996. http://www.ornl.gov/its/archives/mailing-lists/qmail/1996/12/msg00314.html.

[6] K. J. Biba. "Integrity considerations for secure computer systems." Technical Report ESD-TR-76-372, MTR-3153, The MITRE Corporation, USAF Electronic Systems Division, Bedford, MA, April 1977.

[7] M. Bishop and M. Dilger. "Checking for Race Conditions in File Accesses." *Computing Systems*, 9(2):131–152, Spring 1996.

[8] CERT Advisory CA-2000-13. "Two Input Validation Problems in FTPD." 7 July 2000.

[9] CERT Advisory CA-2000-17, "Input Validation Problem in rpc.statd." 18 August 2000.

[10] CERT Incident Note IN-2000-10, "Widespread Exploitation of rpc.statd and wu-ftpd Vulnerabilities." 15 September 2000.

[11] CERT Advisory CA-2000-22. "Input Validation Problems in LPRng." 12 December 2000.

[12] Satish Chandra and Thomas W. Reps. "Physical Type Checking for C." In *Proceedings of the ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, Toulouse, France, September 1999. , pages 66–75.

[13] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Mike Frantzen, and Jamie Lokier. "FormatGuard: Automatic Protection From printf Format String Vulnerabilities." This volume.

[14] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[15] Alan DeKok. "PScan: A limited problem scanner for C source files." Available at http://www.striker.ottawa.on.ca/~aland/pscan.

[16] Martin Elsman, Jeffrey S. Foster, and Alexander Aiken. "Carillon—a System to Find Y2K Problems in C Programs." Available at http://www.cs.berkeley.edu/Research/Aiken/carillon/doc.ps.gz.

[17] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. "Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions." In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, San Diego, CA, October 2000.

[18] David Evans. "Static Detection of Dynamic Memory Errors." *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Philadelphia, Pennsylvania, May 1996, pages 44–53.

[19] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. "A Theory of Type Qualifiers." In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99)*, Atlanta, Georgia, May 1999.

[20] Christopher Harrelson. "Program Analysis Mode." http://www.cs.berkeley.edu/~chrishtr/pam.

[21] Fritz Henglein and Jakob Rehof. "The Complexity of Subtype Entailment for Simple Types." In *Proceedings, Twelfth Annual IEEE Symposium on Logic in Computer Science*, Warsaw, Poland, July 1997, pages 352–361.

[22] Maxime Henrion. "muh IRC bouncer remote vulnerability." FreeBSD Security Advisory FreeBSD-SA-00:57. http://www.securityfocus.com/advisories/2741.

[23] Maxime Henrion. "format string bug in muh." bugtraq mailing list, 09 September 2000. http://www.securityfocus.com/archive/1/81367.

[24] Jarno Huuskonen. "Some possible format string errors." Linux Security Audit Project mailing list, 25 September 2000. http://www2.merton.ox.ac.uk/~security/security-audit-200009/0118.html.

[25] Jarno Huuskonen. "syslog(prio, buf) in mars_nwe." Linux Security Audit Project mailing list, 27 September 2000. http://www2.merton.ox.ac.uk/~security/security-audit-200009/0136.html.

[26] K. Rustan M. Leino and Greg Nelson. "An Extended Static Checker for Modula-3." In Kai Koskimies, editor, *Compiler Construction: 7th International Conference, CC'98*, volume 1383 of Lecture Notes in Computer Science, pages 302-305. Springer, April 1998.

[27] Robert Lemos. "Internet worm squirms into Linux servers." *Special to CNET News.com*, 17 January 2001. http://news.cnet.com/news/0-1003-200-4508359.html.

[28] John C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1(3):245–285, July 1991.

[29] Andrew C. Myers and Barbara Liskov. "Protecting Privacy using the Decentralized Label Model." *ACM Transactions on Software Engineering and Methodology*, 9(4), April 2001.

[30] Tim Newsham. "Format String Attacks." Guardent, Inc. September 2000. `http://www.guardent.com/docs/FormatString.PDF`.

[31] Robert O'Callahan and Daniel Jackson. "Lackwit: Practical Program Understanding With Type Inference." In *Proceedings of the 19th International Conference on Software Engineering*, pp. 338-348, Boston, Massachusetts, May 1997.

[32] Perl Security. `http://www.perl.com/pub/doc/manual/html/pod/perlsec.html`.

[33] Jakob Rehof and Manuel Fähndrich. "Type-Based Flow Analysis: From Polymorphic Subtyping to CFL-Reachability." In *Proceedings of the 28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, London, United Kingdom, January 2001.

[34] Tim J. Robbins. libformat. Available at `http://box3n.gumbynet.org/~fyre/software`.

[35] Pekka Savola. "Very probable remote root vulnerability in cfengine." `bugtraq` mailing list, 1 October 2000. `http://www.securityfocus.com/archive/1/136751`.

[36] Michael Siff, Satish Chandra, Thomas Ball, Thomas Reps, and Krishna Kunchithapadam. "Coping With Type Casts in C." In *ACM Conference on Foundations of Software Engineering (FSE)*, September 1999.

[37] Bjarne Steensgaard. "Points-to Analysis in Almost Linear Time." In *Proceedings of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, January 1996, pages 32–41.

[38] John Viega, J.T. Bloch, Tadayoshi Kohno, and Gary McGraw. "ITS4: A Static Vulnerability Scanner for C and C++ Code." In *16th Annual Computer Security Applications Conference (ACSAC 2000)*, December 2000.

[39] D. Volpano, G. Smith, and C. Irvine. "A sound type system for secure flow analysis." *Journal of Computer Security*, 4(3):1–21, 1996.

[40] D. Volpano and G. Smith. "A type-based approach to program security." *Proceedings of TAPSOFT'97, Colloqium on Formal Approaches in Software Engineering*.

[41] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. "A First Step Toward Automated Detection of Buffer Overrun Vulnerabilities." In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, California, February 2000.

[42] Larry Wall, Tom Christiansen and Jon Orwant. *Programming Perl*, 3rd Edition. July 2000. O'Reilly & Associates.

[43] "WuFTPD: Providing remote root since at least 1994," `bugtraq` mailing list, June 23, 2000, `http://www.securityfocus.com/archive/1/66367`.

A Proof of Theorem 4.1

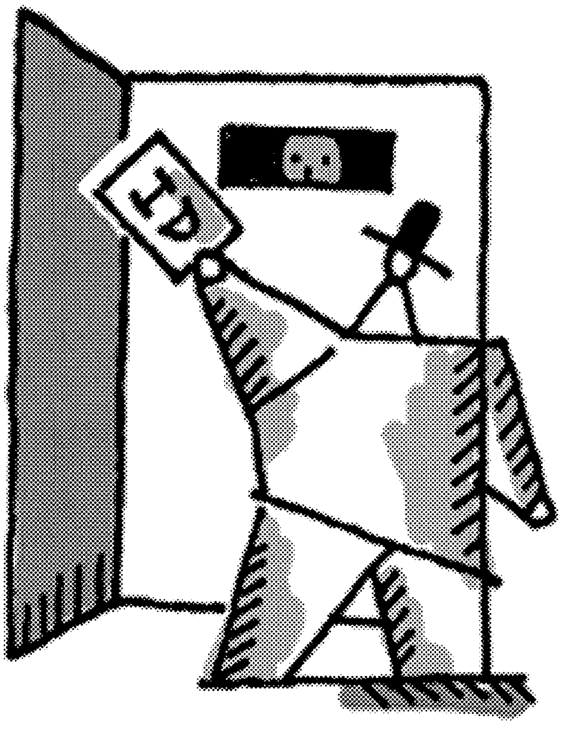
Theorem A.1 *Let $(P, \leq)$ be any finite partial order. Let $(2^{\mathbb{N}}, \subseteq)$ be the lattice of subsets of $\mathbb{N}$ with the set inclusion ordering. Then there exists a mapping $\phi : P \rightarrow 2^{\mathbb{N}}$, such that $\forall x, y \in P, x \leq y \iff \phi(x) \subseteq \phi(y)$ and $\phi(x)$ is a finite subset of $\mathbb{N}$ for all $x \in P$.*

Proof : We prove the theorem by induction on $|P|$.
Base Case : Let $|P| = 1$. Then the claim trivially holds.
Induction Hypothesis : Let the claim hold for all P such that $|P| \leq k$.
Induction Step : $|P| = k + 1$.
Let $(P, \leq)$ be a partial order such that $|P| = k + 1$. Since P is finite, P has a minimal element, say a. Consider the partial order $(P \setminus \{a\}, \leq)$. Clearly this is a partial order and $|P \setminus \{a\}| = k$. Hence by induction hypothesis, there exists $\phi : P \setminus \{a\} \rightarrow 2^{\mathbb{N}}$, such that $\forall x, y \in P \setminus \{a\}, x \leq y \iff \phi(x) \subseteq \phi(y)$ and $\phi(x)$ is a finite subset of $\mathbb{N}$ for all $x \in P \setminus \{a\}$. Let $n = \max_i\{i \in \cup_{x \in P\setminus\{a\}}\phi(x)\}$. Define $\phi' : P \rightarrow 2^{\mathbb{N}}$ as follows.

$$\phi'(x) = \begin{cases} \{n+1\} & \text{if} x = a \\ \phi(x) \cup \{n+1\} & \text{if} x \neq a \text{ and } x \leq a \\ \phi(x) & \text{otherwise} \end{cases}$$

Since a was chosen to be a minimal element, the only relations involving a are of the form $a \leq x$, and for these, by definition, $\phi'(a) = \{n+1\} \subseteq \phi(x) \cup \{n+1\} = \phi'(x)$. For all x such that $a \not\leq x$, we have $\phi'(a) = \{n + 1\} \not\subseteq \phi(x)$ by choice of n. For relations not involving a, the show below that the set containment relations are preserved. Let $\phi(x) \subseteq \phi(y)$. Since $\phi'(y) \supseteq \phi(y)$, the case when $\phi'(x) = \phi(x)$ is trivial. So assume $\phi(x) \subseteq \phi(y)$ and $\phi'(x) = \phi(x) \cup \{n + 1\}$. This implies that $a \leq x$, and $x \leq y$, and therefore $a \leq y$. Thus $\phi'(y)$ would be defined as $\phi(y) \cup \{n + 1\}$, and hence $\phi'(x) \subseteq \phi'(y)$.Thus the induction step holds. ■

AUTHORIZATION



Session Chair: Carl Ellison, *Intel Corporation*

Capability File Names: Separating Authorisation from User Management in an Internet File System

Jude T. Regan* Christian D. Jensen
Department of Computer Science
Trinity College Dublin
juderegan@consultant.com Christian.Jensen@cs.tcd.ie

Abstract

The ability to access and share information over the Internet has introduced the need for new flexible, dynamic and fine-grained access control mechanisms. None of the current mechanisms for sharing information – distributed file systems and the web – offer adequate support for sharing in a large and highly dynamic group of users. Distributed file systems lack the ability to share information with unauthenticated users, and the web lacks fine grained access controls, i.e. the ability to grant individual users access to selected files.

In this paper we present Capability File Names, a new access control mechanism, in which self-certifying file names are used as sparse capabilities that allow a user ubiquitous access to his files and enables him to delegate this right to a dynamic group of remote users. Encoding the capaility in the file name has two major advantages: it is self-supporting and it ensures full compatablity with existing programs.

Capability file names have been implemented in a new file system called CapaFS. CapaFS separates user identification from authorisation, thus allowing users to share selected files with remote users without the intervention of a system administrator. The implementation of CapaFS is described and evaluated in this paper.

1 Introduction

The current suite of Internet protocols and applications [24, 28, 11] allows individuals and organizations convenient access to stored data from every corner of the globe. However, access to such data is typically read-only unless the user is identified on the server on which the data are stored. This prevents users from sharing selected files across organizational boundaries.

A distributed file system allows users read/write access to files stored on remote machines as if they were stored locally. This is a convenient abstraction because it hides the distribution of data from the user in the same way that remote procedure calls hide the distribution of processing. Moreover, most applications accept input from files, which makes distributed file systems the perfect medium for sharing among a heterogenous group of users on the Internet. A distributed file system must implement a flexible and scalable access control mechanism in order to serve different applications. Examples of such applications are mobile users, business-to-business (B2B) commerce and open software development projects.

Most distributed file systems [35, 5, 29, 34, 25, 22, 31, 2] allow users to share files using discretionary access control, which means that the specification of access rights is left to the discretion of the user who "owns" the file. However, in order to allow sharing and enforce access control, the file system generally relies on a local database or password file of users in the system. Moreover, file systems are normally only "exported" to a designated set of machines.

Managing the sets of users that are authorized to access a set of files or the machines that the file system is exported to, is normally a privileged task performed by a small group of system administrators. This means that a third party has to intervene in order for two parties to collaborate. Inflexible local policies, encumbering mechanisms and overworked system administrators are often a major hindrance to collaboration.

As stated above, existing "discretionary" access control

*Jude Regan is currently employed as a Java Developer for Pfizer in Brussels, Belgium.

mechanisms limit the discretion of the user by imposing restrictions on what users are defined locally, what machines files can be exported to and what common authentication framework is required. We wish to extend discretionary access control mechanisms to the Internet, so that users may share their files with any remote user on any remote system without any limitation imposed by local system configuration.

The web provides a convenient medium for sharing of mainly read-only information among a large set of users. Access to information in a particular directory can be protected by passwords [6], thus providing a coarse grained access control mechanism. However, managing several groups with dynamic and overlapping access rights requires the owner to constantly copy, link, or move files among directories; this is not convenient.

We have thus identified the following four properties for a flexible and dynamic access control mechanism for sharing files on the Internet.

No Local Identification Users from different organizations must be able to collaborate, so the access control mechanism cannot rely on user identification in the local domain, e.g., a local data base of registered users.

User Autonomy Collaboration should be immediate, without the delays and hassle of asking a system administrator to define a new user, set up a new group or either export or import a particular file system.

Granularity The current mechanisms for sharing information on the Internet often require the information to be located in particular directories that are "exported" to the remote machines. Different directories may be exported to different sets of machines and with different access rights. Managing these different groups and access rights often requires a complex hierarchy of directories, which makes it difficult to maintain the information. It would be preferred if the user could leave the files where they are and share individual files directly with remote users.

Read/Write Sharing In order to support full collaboration, the access control mechanism must provide equally convenient read and write access to the shared information, although some files could be shared read-only.

Capability file names encode the access rights of the user into the name used by the client to access the file; the file name effectively becomes a capability for that file. The ability to present the capability file name to the file server is enough to allow the user to access the file with the rights encoded in the filename, so no identification is required. We have implemented a proto-type file server, called CapaFS, that acts as a proxy for a particular user and allows him to share his files with everybody on the Internet. The CapaFS server runs entirely in user space, so no intervention is required from the system administrators to set-up or run this service. Each capability file name encodes the access rights for a particular file on the server, this means that individual files are easily shared. The access rights encoded into the file name corresponds to the access rights of the server file system, which normally include both read and write access. An evaluation of the CapaFS proto-type shows that it meets all of the requirements listed above.

The rest of this paper is organized as follows: We start by examining related work on sharing in distributed file systems in section 2. Section 3 describes the principle behind capability filenames. Section 4 describes the design of CapaFS. We evaluate the implementation of CapaFS in Section 5. Some directions for future work are outlined in Section 6 and our conclusions are presented in Section 7.

2 Related Work

In the following, we examine a number of networked or distributed file systems that address some of the issues that we are trying to solve.

We present a short survey of existing access control mechanisms, before examining the distributed file systems.

2.1 Access Control Mechanisms

The access rights of all principals in the system are normally encoded in an access control matrix [20], either in the form of access control lists (ACLs) or capabilities.

An ACL is associated with every resource in the system. It lists all the users who are authorized to access the resource along with their access rights. Strong authentication mechanisms, such as Kerberos [18] or X.509 [27], allow ACLs to be used in a networked environment. The ACL relies on the authenticated identity of the user, which requires the identity of the user to be known before access can be granted. Moreover, ACLs do not scale very well, because it is difficult to delegate the right to delegate, i.e., the right to modify the ACL. This means that ACLs in large organizations have to be organized into a hierarchy where the right to delegate access rights, within a part of the organization, corresponds to the right to modify the ACL in the corresponding branch of the ACL hierarchy. Thus, the hierarchy of ACLs mustreflect the structure of the organization in which they are used. [9]

A capability is an unforgeable token that identifies a resource and lists the access rights granted to the holder of the capability. Anyone holding a copy of the capability may access the resource with the access rights specified by the capability. Capabilities can be stored in data structures in user space and copied or transferred among users. This makes it easy for users to create new adhoc work groups and to distribute access rights to that group. Because knowledge of a capability conveys right to access the specified object or file, capabilities must be protected from theft and disclosure, e.g., by encrypting part of the capability.

2.2 Amoeba File Server

Amoeba is a distributed object oriented operating system designed for a network of closely connected machines. Amoeba uses sparse capabilities to protect all objects in the system including files [38]. Capabilities are stored in data structures in a process' address space and can be exchanged freely among processes.

The Amoeba file service consists of two distinct servers: the directory server and the Bullet file server [32]. The directory server maps human-chosen ASCII names to capabilities used by the system. The Bullet file server implements the required functionality to create, read and write files. The create operation returns a capability that must be used by subsequent read and write operations. This capability can be stored in the directory service for later retrieval.

The sparse capability model implemented by the Amoeba file service has inspired the access control model used in capability file names. An important difference between the two systems is that capability file names are designed to ensure compatibility with existing applications, while Amoeba is not.

2.3 NFS

In NFS [34, 37], the server trusts the identification performed by the client. Clients and servers coordinate their user identifiers. File systems are explicitly exported to a designated set of clients. In order to access a file on the server, the user must be defined as a user on the server machine and the client machine must be added to the export list of the server. Both of these operations are privileged, i.e., require the intervention of a system administrator.

2.4 AFS

AFS [16, 35, 36, 10] is probably the most widely used wide-area file system. AFS mounts all remote file systems under a single directory (`/afs`). The set of remote directories available under the global mount point is managed locally and changes require the privileges of a system administrator.

AFS uses Kerberos [18] to authenticate users, which penalizes use across administrative boundaries. To facilitate collaboration, users of such systems form inconveniently large administrative realms, so anyone they may want to collaborate with will be in the same realm. Kerberos authentication suffers from this problem [4]. System administrators responsible for setting up user accounts often could not do so without the intervention of the Kerberos administrator. Administrators of AFS client machines must enumerate every single file server the client can talk to. A user of an AFS client cannot access a server the administrator did not include.

2.5 SFS

SFS [12, 22, 21] is a global decentralized file system. Like AFS it uses a global mount point to provide a single name space across all machines in the world while avoiding centralized control. Public-key cryptography is used to authenticate all entities in the system.

SFS introduces self-certifying pathnames to name files in this global name space. A self-certifying pathname consists of the file server's *location*, e.g., a host name or an IP address, and a *HostID* that tells the client how to certify a secure channel to the server.[1] The self-

[1]The HostID is actually a cryptographic hash of the server's location and its public key.

	No Local Identification	User Autonomy	Granularity	Read/Write Sharing
Amoeba	YES	YES	FILE	YES
NFS	NO	NO	FILE	YES
AFS	NO	NO	DIRECTORY	YES
SFS	NO	NO	FILE	YES
Truffles	NO	YES/NO	VOLUME	YES
WebFS	NO	YES/NO	FILE	YES

Table 1: Comparison of the surveyed file systems

certifying pathname entirely suffices to name and certify the file server.

SFS relies on a trusted third party to authenticate the user and the client machine. This means that collaboration is only possible if client and server have a common root for their certification authorities.

It is important to note that SFS self-certifying file names are used for authentication, not for authorization. Access control in SFS relies on user and group IDs, so the intervention of a system administrator with special privileges is required to create an account before a remote user can access files on the server. The authentication framework introduced with self-certifying file names is complementary to the authorization framework introduced with capability file names.

2.6 Truffles

Truffles [31] is a distributed file system that attempts to make file sharing between users in different administrative domains both simple and secure. It uses the Ficus file system [15] to offer replication and sharing of files and Privacy Enhanced Mail (PEM) as a secure transport mechanism. Truffles allows users to share volumes (subsets of the entire file system) with little intervention from the system administrators once the volumes have been defined. Ficus replicates volumes using an optimistic one-copy policy and all systems sharing the volume may hold their own replica. However, file sharing relationships and file data transfer rely on Privacy Enhanced Mail (PEM) which requires a nonempty intersection between the public key infrastructures that each user belongs to; the slow adaptation of secure HTTP shows how this limits collaboration among users.

2.7 WebFS

WebFS [39, 2] is a global file system that uses HTTP [11] as the transport protocol between client and file server. The advantage of this approach is that existing URLs can be used as file names and accessed through the file system.

Authentication of both clients and servers are based on X.509 certificates [3]. WebFS maintains an ACL for each file consisting of the X.509 certificate and permissions for each authorised user. Users may therefore share their files with anyone who has a certificate *from a certification authority (CA) trusted by the local domain.* Managing the set of trusted CAs requires special privileges. Thus, authorisation in WebFS relies on the hierarchy of certification authorities, which places the users of WebFS under the control of these certification authorities.

2.8 Summary

A summary the surveyed systems is presented in Table 1.

NFS and AFS are very similar. System administrators are required to enumerate all exported file systems and all machines with remote access. With NFS, all machines form part of the trusted computing base, while AFS supports sharing among machines in different administrative domains. SFS allows free choice of authentication mechanism, but authorization relies on local ACLs. Both Truffles and WebFS allows dynamic sharing of files among users who trust the same certification authority, but it is impossible to share files with users without a recognized certificate. All the surveyed file systems provide read/write access and fine granularity, so they appear not to be real issues. However, one of the most popular media for information sharing, the web, provides little support for either.

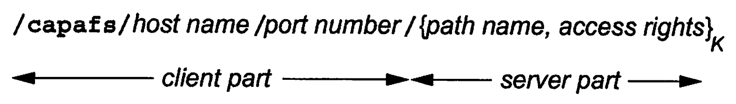


Figure 1: Basic capability file name

3 Capability File Names

The goal of capability file names is to allow users belonging to one or more organizations, to set up ad-hoc work groups without limitations imposed by the system or the intervention of their system administrators. Each user in the group should be allowed to share selected files with other members of the group, without compromising his remaining files, nor the files of other users on his system.

3.1 Basic Capability File Names

The basic capability file name consists of two parts, the "client part" that allows the client machine to identify the server and the "server part" which allows the server to identify the file and encodes the access rights. The server part is encrypted by the server to protect it from tampering. The structure of a capability file name is shown in Figure 1.

The client part of the capability file name consists of a prefix ("`/capafs/`") that identifies it as a capability file name. It also contains the host name of the machine running the server and the port number used by the server.

The server part consists of the absolute path name of the file on the server and the permissions (`-rwx`) granted to the client presenting the capability filename. The server part is encrypted (with key K) to protect it from tampering.

This design is similar to the sparse capabilities used in Amoeba (cf. 2.2). However, sparse capabilities are physical objects that are supported by the system and manipulated explicitly by programs. Capability files names are virtual objects that require no special support from the underlying system and they are manipulated as ordinary file names by programs. This difference is best illustrated by analogy with plane tickets. Traditional capabilities are equivalent to paper tickets (data structures) that have to be acquired and presented at the check-in desk (the file server) in order to grant access to the flight (the file). Capability file names are equivalent to confirmation numbers used when tickets are bought on the Web. Knowledge of a valid confirmation number (capability file name) is proven to the check-in desk in order to grant access to the flight.[2]

3.1.1 Creation of a Capability File Name

Capability file names are created on the server using a separate program. This program takes the host name and the port number that the server is using, the path name of the file and the permissions to be encoded into the capability file name as parameters. It then reads the encryption key from a file stored in the user's home directory and encrypts the server part with this key. It then creates a string by concatenating the prefix with the host name, the port number and the server part and returns it in a string to the remote user. Only the server needs the ability to decrypt the server part of the capability file name, so a fast symmetric cipher may be used. However, a number of possible extensions rely on public-key cryptography, so the server part could also be encrypted using the server's private-key. This reduces the number of keys that the server has to maintain, but it also reveals the contents of the server part of the capability file name.

3.1.2 Using Capability File Names

Each user, who wishes to share his files, must start a server to act as a proxy for remote file operations. This server is described in greater detail in section 4.4.

In order to prevent disclosure of the capability file names and to ensure the confidentiality and integrity of transferred file data, all communication between client and server must use a secure channel. This channel must encrypt all communication, but need not authenticate the end-points to each other, e.g., a fast symmetric encryption algorithm may be used with the Diffie-Hellman key exchange [7]. This solution is vulnerable to the man-in-the-middle attack, an extension that solves this problem is proposed in Section 3.2.

[2]This analogy breaks if two people present the same confirmation number to the check-in desk. The file server grants both users access to the file, while only one person can physically board the plane.

3.1.3 Delegation of a Capability File Name

Secure delegation of capability file names is orthogonal to the mechanism described in this paper, however it is important that the capability file name is protected from disclosure while in transit.

3.1.4 Persistence of a Capability File Name

Capability file names are not persistent in themselves; they are simply names (i.e., character strings) that are lost when the client terminates or if the client fails; server failure does not affect the validity of a capability file name. However, a capability file name can be stored on stable storage or serve as the source of a symbolic link, thus making it persistent. Using a symbolic link to point to a capability file name allows its holder to assign a meaningful name to the remote file, although this name only has local significance.

3.1.5 Revocation of a Capability File Name

In order to allow revocation of capability file names, the server must maintain a capability revocation list (CRL) of all capability file names that have been revoked. The CRL grows with time and searching through it may become prohibitive. One solution is to limit the time that a capability file name is valid (i.e., include a timeout value in the server part). As the timeout is only used on the server, client and server clocks do not have to be synchronized. However, new versions of the capability file names must be acquired when the capability file name expires.

Another solution is to mark files that have a revoked capability file name associated with them (e.g., changing the files meta-data, such as the inode number), the CRL is only searched if the file is marked. The CRL is still needed because there may be number of valid operations that change the meta data, e.g., restoring a file from backups changes the inode number.

3.2 Capability File Names with Server Authentication

The secure communication channel discussed in the basic scheme above suffers from the man-in-the-middle attack, where a third party intercepts the initial message from the client and sets up connections to both client and server. The man-in-the-middle relays all messages between client and server and knows both keys. We therefore need an authentication mechanism that escapes the problems of centralized control. In order to authenticate the server, its public-key (SPuK) is added to the client part of the capability file name. In this case the server's private-key (SPrK) is used to encrypt the server part of the capability file name. The structure of a capability file name with server authentication is shown in Figure 2(a).

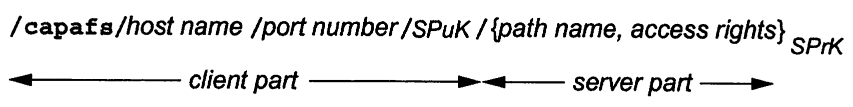
The server is authenticated in the following way before the client sends the capability file name to the server. The client selects a session key to be used by the secure channel. The session key is encrypted with the server's public-key and sent to the server. The server responds with a message encoded with the session key, which proves its possession of the server's secret key and thereby authenticates the server. The client is implicitly authenticated as belonging to the set of authorized user, when the server receives the capability file name. This authentication protocol is very similar to the protocol used in SFS [23].

3.3 Capability File Names with Client Authentication

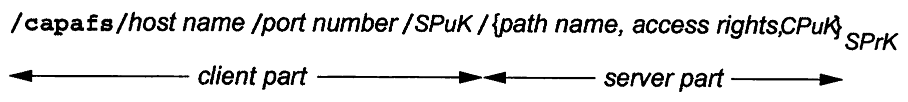
As mentioned above, the client is implicitly authenticated through the possession of the capability file name. However, knowing the identity of the client "enables the monitoring, mediating, and recording of capability propagations to enforce security policies including the $\star$-property in the Bell-LaPadula model" [14]. Moreover, it introduces accountability into the system. The structure of a capability file name with client (and server) authentication is shown in Figure 2(b).

The identity of the client is used on the server side and should be included in the server part *before* the capability file name is given to the client.

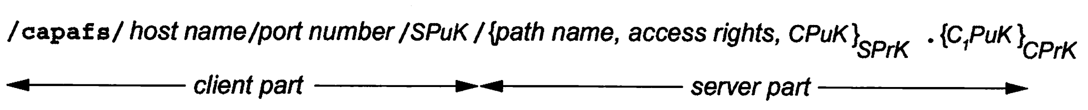
The authentication of the client follows the scheme described in Section 3.2. After the client has received the first message with the session key from the server, it signs the capability file name with his private-key and sends it to the server. It is important to note that the authentication of the client does not necessarily depend on his physical identity; the key-pair used could be created explicitly for this capability file name.


/capafs/host name /port number/SPuK/{path name, access rights} SPrK
client part
server part

(a) Capability file name with server authentication


/capafs/host name /port number/SPuK/{path name, access rights,CPuK} SPrK
client part
server part

(b) Capability file name with client authentication


/capafs/ host name/port number/SPuK/{path name, access rights, CPuK}SPrK . {C1PuK}CPrK
client part
server part

(c) Capability file name propagation chain

Figure 2: Extensions to the basic capability file names

3.4 Capability File Names with Propagation Limitation

The introduction of client authentication above, allows us to monitor the use of capability file names. The addition of delegation chains [1], allows the server to impose restrictions on the further delegation of a capability file name, e.g., restricting further delegation to a known set of recipients, restricting the right to delegate the capability file name to the original recipient or preventing delegation altogether. Moreover, it allows the server to implement different delegation policies for different files. This approach is similar to transfer certificates in CRISIS [3] or authorization proxies implemented on top of Kerberos [26].

The structure of a capability file name with propagation limitation is shown in Figure 2(c).

A user (C) who wish to further delegate a capability file name to another user (C_1), encodes the public-key of the recipient (C_1 PuK) with his own private-key (CPrK) and appends it as an extension to the filename. Further delegation of this capability file name will add another extension, so a long delegation chain means that the filename will have many extensions. When the server receives the delegated capability filename, it first retrieves the public-key of the original recipient (CPuK). It uses this key to decrypt the public-key included in the first extension, this process is repeated until the final public-key is retrieved. This final key is then used to authenticate the requesting client, as described in Section 3.2.

3.5 Summary

Capability file names allow flexible and dynamic sharing of files among users in different organizations, without the intervention of system administrators in either organization. A number of extensions to the basic capability file name allow clients and servers to authenticate each other. This authentication does not rely on digital certificates to prove the physical identity of either party. Instead, the knowledge of a private-key is used to authenticate users. This allows collaboration without compromising the semi-anonymous nature of the Internet, i.e., a user may assume a virtual identity and associate it with a public/private key pair. The user may then use a grassroots mechanism, such as the PGP web of trust, to distribute the public-key associated with this virtual identity.

4 CapaFS

CapaFS [30] is a userlevel file system that implements basic capability file names. CapaFS uses AES [17, 8] for symmetric encryption and RSA [33] for asymmetric encryption of the server part of the capability file name. AES is also used to establish a secure channel between client and server. We currently use PGP [13] to provide confidentiality and authentication of capability file names distributed per email, i.e., we do not require a traditional hierarchical public key infrastructure.

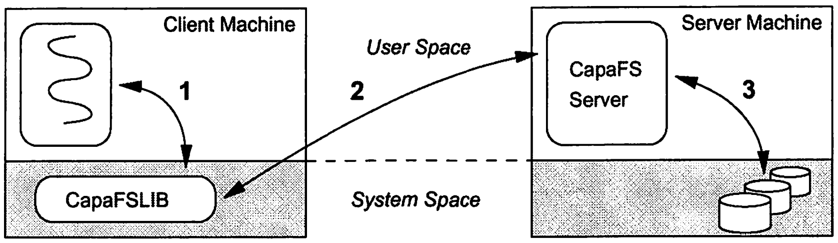


Figure 3: The CapaFS architecture

4.1 Overview

CapaFS consists of two parts: a shared library that replaces `libc` on the client machine and a user level file server. The architecture of CapaFS is illustrated in Figure 3.

The library intercepts all operations on files under `/capafs` (1) and redirects them to the designated server (2). If the operation is `open()`, the server verifies the capability file name and returns a file handle or an error to the client. All other operations use this file handle to access the file, [3] so the server only has to decrypt the secret part of the capability file name once. Clients with valid file handles can perform remote file operations using standard NFS semantics (3).

4.2 Creating Capability File Names

Two programs are used to create a capability file name: `CapaFSKeys` creates an encryption key, either a symmetric key or the private-key of a public/private key pair, and `CapaFSFile` produces a capability file name from the file's location, the server's port number, the remote user's access rights, and the encryption key created by `CapaFSKeys`.

It is important to note that CapaFS is not a distributed file system, i.e., all files are created locally, it simply provides remote users with access to locally stored files. The symbolic link mechanism described in Section 3.1.4 can be used to create the illusion of local and remote files stored in the same directory.

[3]The file handle is a large randomly chosen integer, thus effectively a sparse capability.

4.3 Capability File System Wrapper Library

The CapaFS shared library CapaFSLIB is used by a client to add the necessary functionality to handle CapaFS capability filename to the operating system. The CapaFS shared library replaces the standard shared C library, `libc`. The following file operations are wrapped: `open`, `close`, `read`, `write`, `lseek` and `fcntl`.

CapaFSLIB can be installed in any directory as long as the shared library loader knows where to find it; this is normally achieved by setting a variable in the user's environment. This means that Capability File Names can be installed without any intervention from the system administrators, thus promoting user-to-user collaboration.

4.4 Capability File Server

CapaFS servers can be set up easily without any system administrator intervention. The CapaFS server runs in user space and can be started by any user in the system. The server runs with the privileges of the user who starts it to ensure that the server is restricted to only the files that this user can access. This means the server has access to exactly those services and files that are necessary for its operation, following the principle of least privilege. The underlying operating system enforces this restriction. This maintains the integrity of the system, and ensures a CapaFS server cannot be hijacked to gain access to another user's or system files, thus providing no extra threat to systems.

A user who wants to share his files with others simply starts the CapaFS server. Once a user has created capability filenames and given them to parties they trust, they can start a CapaFS server which will handle requests from remote clients and allow sharing of the specified files with the allowed permissions.

4.5 Granularity

The CapaFS server effectively acts as a proxy between the remote user and the local file system. Capability file names can be used to grant access to both individual files, and to all files in a directory.

Standard file system semantics apply to the remote operations on both files and directories with the following exceptions: file operations are executed with the intersection of the access rights of the user who started the server and the permissions encoded in the capability file name, and the right to access a file in a particular directory implies the right to search all higher level directories.

5 Evaluation

In the following we present an evaluation of the functionality, security and performance of the developed prototype.

5.1 Functionality

The functionality of CapaFS has been evaluated through the following scenario. Two researchers are writing a paper for a scientific conference; one author is located in Belgium, the other in Ireland. The security policies of their respective organizations prevent either from easily obtaining an account on the other's system.

With CapaFS, any user can start a server on a machine with access to the Internet, thus allowing him to share his files with anybody without the intervention of the system administrator of his machine.

They decide to format the paper with the LaTeX [19] text preparation system. The first author writes an initial draft of the paper, creating all the required LaTeX source files in the process; a listing of the source directory is shown in Figure 4(a).[4] He then starts the CapaFS server, creates capability filenames, with read/write permission, for all LaTeX files and sends them to the second author over a secure channel.

[4]Non-essential details have been deleted from the directory listings. In order to protect the local system, the host name and port number have been changed and the server part of the capability file name is truncated after 8 characters.

The second author then creates a new local directory and symbolic links in that directory that point to the received capability file names. A listing of the second author's source directory is shown in Figure 4(b). The second author can now edit the source files directly and process them with LaTeX locally.

A separate capability file name is used for each file – instead of a directory capability file name – to improve the speed of formatting the document. A directory capability file name would mean that temporary files created by LaTeX would be created on the server. Instead, these files are created locally. The directory of the second author – after processing the source files with LaTeX – is shown in Figure 4(c).

In our example one author created all the files and distributed the required capability filenames to the other, but the system allows both users to start CapaFS servers and thus both users to create new files.

5.2 Security

We assume that an attacker has control over the network between client and server, but that the security of neither client nor server machine has been compromised.

An attacker who controls the network may attack a system by interception, interruption, modification and fabrication of messages.

Interception An eavesdropper reads the message as it passes on the network, thus compromising the confidentiality of the message. CapaFS ensures confidentiality by encrypting the communication between client and server.

Interruption Preventing the message from reaching its destination results in denial of service. This type of attack cannot be prevented without complete control over the network.

Fabrication Creation of new messages allows an attacker to masquerade as either client or server. Masquerading as the client requires him to know a capability file name, but they are only sent over secure channels which means that they cannot be known by outsiders. Masquerading as the server allows the attacker to learn capability file names as clients connect to it and send erroneous data to the client. In Section 3.2 we proposed a mechanism similar to the self certifying file names in SFS;

```
-rw-r--r--  [...deleted...] paper.bib
-rw-r--r--  [...deleted...] paper.tex
```

(a) Directory on the server

```
lrwxr-xr-x  [...deleted...] paper.bib -> /capafs/fs.dsg.cs.tcd.ie/9999/5be34dd[...deleted...]
lrwxr-xr-x  [...deleted...] paper.tex -> /capafs/fs.dsg.cs.tcd.ie/9999/715a9f3[...deleted...]
```

(b) Directory on the client

```
-rw-r--r--  [...deleted...] paper.aux
-rw-r--r--  [...deleted...] paper.bbl
lrwxr-xr-x  [...deleted...] paper.bib -> /capafs/fs.dsg.cs.tcd.ie/9999/5be34dd[...deleted...]
-rw-r--r--  [...deleted...] paper.blg
-rw-r--r--  [...deleted...] paper.dvi
-rw-r--r--  [...deleted...] paper.log
lrwxr-xr-x  [...deleted...] paper.tex -> /capafs/fs.dsg.cs.tcd.ie/9999/715a9f3[...deleted...]
```

(c) Directory on the client after processing the files

Figure 4: Typesetting a paper on a remote machine

this mechanism is not yet implemented. Adding the servers public key to the client part of the capability file name allows the client to authenticate the server.

Modification The integrity of file data is compromised if the message is modified between the client and the server. However, modified messages cannot go undetected because the client and the server communicate over an encrypted link.

Encryption of communication between client and server ensures confidentiality and integrity. Clients are explicitly not authenticated, because requests may arrive from any node, and knowledge of the capability file name is enough to grant access to the file. Authentication of the server is easily achieved by including the public key of the server in the capability file name as described in Section 3.2.

5.3 Performance

Performance was not of primary concern to us, but the capability file name mechanism is unlikely to be widely adopted if it introduces a very large overhead. We therefore decide to evaluate the performance of CapaFS. The performance of CapaFS is compared to NFS, a widely used networked file system. However, it is important to note that CapaFS is designed to provide functionality which NFS cannot provide, so the comparison should only be taken as an indication of the usability of CapaFS.

There are several factors, which might differentiate CapaFS performance from NFS performance. First of all CapaFS runs entirely in userspace, while NFS has been integrated into the operating system kernel. Second, CapaFS is more CPU intensive than NFS because of the RSA public key encryption and integrity checks performed on CapaFS capability filenames. The open operation requires the user-level server to decrypt the server part of the capability file name, in order to reveal the filename and permissions. The overhead of decrypting the capability is proportional to the bit-size of the encryption used. Finally, NFS has been fine-tuned for more than a decade, while CapaFS is a recent prototype implementation, e.g., none of the caching strategies used by NFS have currently been implemented in CapaFS.

The performance of CapaFS and NFS is now compared and discussed. The tests of both CapaFS and NFS were performed using two 1GHz Pentium III machines running RedHat Linux with 256MB of memory. The two machines were connected with 100Mbit Ethernet through a 100Mbit switch.

The client and server were set up on different machines running in standard multi-user mode. The tests covered the following operations for both CapaFS and NFS: Opening or lookup of a remote file, reading a 1KB from a remote file, and writing a 1KB to a remote file.[5]

All tests are timed from the point the operation is invoked, until the point when a result is returned. In addition to this, all measurements are performed on the client side, accessing files on the server side as usual. The performance results of these tests are given in Table 2.

File operation	CapaFS	NFS
`open()`	1292 μs	159 μs
`read()`	117 μs	94 μs
`write()`	987 μs	8 μs

Table 2: Performance comparison of CapaFS and NFS

Our measurements show that CapaFS has an acceptable absolute performance, the most expensive operation (`open()`) costs little over a millisecond, so the cost of file system operations are dominated by communications costs in a wide area network.

CapaFS takes significantly longer than NFS to open a file (x10), but it is only called once when the file is initially opened (subsequent read and write operations use a file handle returned by the open call). The higher cost of the open command is to be expected, because the server part of the capability file name has to be decrypted. The cost of reading data is roughly equivalent in the two systems, while the cost of writing data to a remote file is significantly higher in CapaFS (x100). We attribute the big difference in write performance of NFS and CapaFS to NFS's use of asynchronous writes, which makes NFS significantly faster when writing data to a remote file. The asynchronous write strategy is acceptable on local area networks, where the probability of errors and partitions is low, but we do not believe that such optimizations are appropriate in a file system designed for use in wide area networks.

5.4 Summary

We have shown that CapaFS meets all of the requirements defined in Section 1.

No Identification CapaFS allows dynamic sharing of selected files, without identification of the remote user; the knowledge of the capability file name is enough to grant access. Neither of the users described in Section 5.1 holds an account on the other user's machine.

No Intervention Both CapaFSLIB and the CapaFS server runs in user space and were setup without the intervention of the system administrator. However, if every user is to run a CapaFS server, some support from system administrators would be needed to coordinate the local use of port numbers. We didn't experience problems with port number allocation during these experiments.

Fine Granularity Capability file names can be used to grant access to individual files, as well as directories.

Read/Write Access The measurements presented in Section 5.3 successfully read and wrote files across the Internet.

[5] Link-level encryption has been disabled in CapaFS in order to provide comparability with NFS.

6 Future Work

The current implementation of CapaFS relies on a wrapper library on the client's machine. We would like to extend this with a file system implemented as a loadable kernel module. This allows us to implement efficient caching policies for remote files and decreases the overhead of context-switching between the user library and the kernel. Installation of the loadable kernel module for the file system requires the intervention of the system administrator, but this installation can be performed once for all users.

The propagation limitation mechanism outlined in Section 3.4 should also be implemented. This reduces the risk of sharing files by limiting the number of users authorized to delegate the capability file names.

7 Conclusions

In this paper we addressed the issue of sharing information in large open (Internet) distributed file systems.

We presented a new access control mechanism designed to facilitate sharing among dynamic groups of non-authenticated users. This design is implemented in CapaFS, a global and decentralized file system that allows users to collaborate with other users regardless of location and with no prior arrangements or intervention by system administrators. The system uses filenames as sparse capabilities to name and grant access to files on remote servers. Users can share files without the intervention of system administrators, by exchanging capability filenames with parties that they trust. Unlike other systems, CapaFS has no need to authenticate the client machine to the server. A client must simply prove possession of a valid capability filename; this is necessary and sufficient proof of authority to perform the operations – encoded in the capability – on the file it names. CapaFS does not need to establish trust between client and server, it only needs to verify the validity of the CapaFS filename.

Capability file names may be used successfully in many different environments to provide previously unavailable functionality. Roaming mobile users who share files from their home site with the people they are visiting is one setting. CapaFS may also be used in large businesses, to cross administrative boundaries or company boundaries in a virtual enterprise. People who work with semi-anonymous users over the Internet and collaborate on projects, may use CapaFS to facilitate and promote sharing.

A decentralized file system with global authentication and flexible authorization can free users from many of the problems that have developed due to increased security and centralized control.

Acknowledgements

The authors would like to thank our colleague Stefan Weber for his help with the performance evaluation presented in this paper, and also the paper's anonymous reviewers for their comments, which have helped us to improve the paper.

References

[1] T. Aura. Distributed access-rights management with delegation certificates. In J. Vitek and C.D. Jensen, editors, *Secure Internet Programming*, number 1603 in Lecture Notes in Computer Science LNCS, pages 211–235. Springer Verlag, 1999.

[2] E. Belani, A. Thornton, and M. Zhou. Authentication and security in WebFS, January 1997.

[3] E. Belani, A. Vahdat, T. Anderson, and M. Dahlin. The crisis wide area security architecture. In *Proceedings of the 7th USENIX Security Symposium*, pages 15–29, San Antonio, Texas, U.S.A., January 1998.

[4] S. M. Bellovin and M. Merrit. Limitations of the Kerberos authentication system. *Computer Communications Review*, 20(5):119–132, October 1990.

[5] A. Birrell, A. Hisgen, C. Jerian, T. Mann, and G. Swart. The Echo distributed file system. Technical Report 111, Digital Equipment Corp. Systems Research Center, 1993.

[6] K. Coar. *Using .htaccess Files with Apache*, 2000.

[7] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, November 1976.

[8] Federal Information Processing Standard *Draft. Advanced Encryption Standard (AES)*. National Institute of Standards and Technology, 2001.

[9] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI certificate theory. Technical Report 2693, Network Working Group, IETF, September 1999.

[10] C. F. Everhart. Conventions for names in the service directory in the AFS distributed file system. Technical report, Transarc Corporation, March 1990.

[11] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. Request for Comments (RFC) 2616, Network Working Group, IETF, 1999.

[12] K. Fu, M. F. Kaashoek, and D. Mazières. Fast and secure distributed read-only file system. In *Proceedigs of the 4th Symposium on Operating Systems Design and Implementation*, pages 181–196, San Diego, California, U.S.A., October 2000.

[13] S. Garfinkel. *PGP: Pretty Good Privacy*. O'Reilly & Associates, Inc., 1994.

[14] L. Gong. A secure identity–based capability system. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 56–63, Oakland, California, U.S.A., May 1989.

[15] R. G. Guy, J. S. Heidemann, W. Mak, T. W. Page Jr., G.J.Popek, and D. Rothmeier. Implementation of the Ficus replicated file system. In *Proceedings of the Summer USENIX Conference*, pages 63–71, June 1990.

[16] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, m. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, 1988.

[17] V. Rijmen J. Daemen. The block cipher rijndael. In J.-J. Quisquater and B. Schneier, editors, *Smart Card Research and Applications*, Lecture Notes in Computer Science (LNCS) 1820, pages 288–296. Springer-Verlag, 2000.

[18] J. Kohl and C. Neuman. The Kerberos network authentication service (v5). Request for Comments (RFC) 1510, Network Working Group, IETF, September 1993.

[19] L. Lamport. *LaTeX – A Document Preparation System – User's Guide*. Addison-Wesley, 1985.

[20] B. W. Lampson. Protection. In *Proceedings of the 5th Princeton Symposium on Information Sciences and Systems*, pages 437–443, March 1971. reprinted in *Operating Systems Review, 8, 1* January 1974 pages 18–24.

[21] D. Mazières. Security and decentralised control in the SFS distributed file system. Master's thesis, MIT Laboratory of Computer Science, 1997.

[22] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *Proceedings of the 17th Symposium on Operating Systems Principles*, pages 124–139, Kiawah Island, S.C., U.S.A., 1999.

[23] David Mazières and M. Frans Kaashoek. Escaping the evils of centralized control with self-certifying pathnames. In *Proceedings of the 8th ACM SIGOPS European workshop: Support for composing distributed applications*, pages 118–125, Sintra, Portugal, September 1998.

[24] N. J. Neigus. File transfer protocol for the ARPA network. Request for Comments (RFC) 542, Bolt Beranek and Newman, Inc., August 1973.

[25] B. C. Neuman. Prospero: A tool for organizing internet resources. *Electronic Networking: Research, Applications and Policy*, 5(4):30–37, 1992.

[26] B. C. Neuman. Proxy-based authorization and accounting for distributed systems. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 283–291, Pittsburgh, Pennsylvania, U.S.A., May 1993.

[27] Telecommunication Standardization Sector of ITU. *Information Technology — Opens Systems Interconnection — The Directory: Authentication Framework*. Number X.509 in ITU–T Recomandation. International Telecomunication Union, November 1993. Standard international ISO/IEC 9594–8 : 1995 (E).

[28] J. B. Postel. Simple mail transfer protocol. Request for Comments (RFC) 821, Information Sciences Institute, University of Southern California, August 1982.

[29] H. C. Rao and L. L. Peterson. Accessing files in an internet: The JADE file system. *IEEE Transactions on Software Engineering*, 19(6):613–624, June 1993.

[30] J. Regan. Capafs: A globally accessible file system. Department Technical Report TCD-CS-1999-70, Department of Computer Science, Trinity College Dublin, 1999.

[31] P. Reiner, T. Page Jr., G. Popek, J. Cook, and S. Crocker. Truffles – a secure service for widespread file sharing. In *Proceedings of the Privacy and Security Research Group Workshop on Network and Distributed System Security*, 1994.

[32] R. van Renesse, A. S. Tanenbaum, and A. Wilschut. The design of a high-performance file server. In *Proceedings of the 9th International Conference on Distributed Computing Systems*, pages 22–27, Newport Beach, california, U.S.A., June 1989.

[33] R. L. Rivest, A. Shamir, and L. Adleman. On a method for obtaining digital signatures and public key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[34] R. Sandberg, D. Goldberg, Kleinman S, D. Walsh, and B. Lyon. Design and implementation of the Sun Network File System. In *Proceedings of the Summer 1985 USENIX Conference*, pages 119–130, Portland, Oregon, U.S.A., June 1985.

[35] M. Satyanarayanan. Integrating security in a large distributed system. *ACM Transactions on Computer Systems*, 7(3):247–280, 1989.

[36] M. Satyanarayanan. Scalable, secure and highly available file access in a distributed workstation environment. *IEEE Computer*, pages 9–21, May 1990.

[37] Sun Microsystems Inc. NFS: Network file system protocol specification. Request for Comments (RFC) 1094, Network Working Group, March 1989.

[38] A. S. Tanenbaum, S. J. Mullender, and R. van Renesse. Using sparse capabilities in a distributed operating system. In *Proceedings of the 6th International Conference in Computing Systems*, pages 558–563, June 1986.

[39] A. Vahdat, P. Eastham, and T. Anderson. Webfs: A global cache coherent file system. Department of Computer Science, UC Berkeley, Technical Draft, 1996.

Kerberized Credential Translation: A Solution to Web Access Control

Olga Kornievskaia
Peter Honeyman
Bill Doster
Kevin Coffman

Center for Information Technology Integration
University of Michigan
Ann Arbor

{aglo,honey,billdo,kwc}@citi.umich.edu

Abstract

Kerberos, a widely used network authentication mechanism, is integrated into numerous applications: UNIX and Windows 2000 login, AFS, Telnet, and SSH to name a few. Yet, Web applications rely on SSL to establish authenticated and secure connections. SSL provides strong authentication by using certificates and public key challenge response authentication. The expansion of the Internet requires each system to leverage the strength of the other, which suggests the importance of interoperability between them.

This paper describes the design, implementation, and performance of a system that provides controlled access to Kerberized services through a browser. This system provides a single sign-on that produces both Kerberos and public key credentials. The Web server uses a plugin that translates public key credentials to Kerberos credentials. The Web server's subsequent authenticated actions taken on a user's behalf are limited in time and scope. Performance measurements show how the overhead introduced by credential translation is amortized over the login session.

1 Introduction

Access control for Web space is often viewed in terms of gating access to Web pages where the job of the Web server is limited to simple file reads. The functionality provided by Web servers has grown considerably making it the most popular technology on the Internet. With the expansion of the Internet, many new kinds of services are accessible from the Web, increasing Web servers' importance and scope. For example, a Web server may serve information stored in backend databases. A Web interface to backend services is considered to be more user-friendly and accessible compared to predominant text-based interfaces.

The possibilities opened by the use of a Web server to access a variety of backend services pose challenging questions on how to retain access control of backend services. A Web server could potentially become another access control decision point, increasing the burden on the server and its administrators. It would have to comply with the same security requirements as all of the backend services it fronts, increasing its potential as a place for system compromise.

A solution that provides end-to-end authorization would allow the end service to retain control over the authorization decisions. Furthermore, it would obviate constructing and maintaining consistent replicas of authorization policies.

In practice, authorization mechanisms are tied to authentication mechanism: end-to-end authorization requires end-to-end authentication. A mismatch in authentication mechanisms prevents a Web server from using authorization mechanisms provided by backend servers. While Web servers support SSL authentication with certificates, this does not provide credentials for access to AFS file servers, LDAP directory servers, and KPOP/IMAP mail servers, which use Kerberos for client authentication. To provide end-to-end authorization, we address the problem of end-to-end authentication.

We motivate the end-to-end authentication problem by considering the following scenario:

Alice attends the University of Michigan, where she enjoys access to a variety of computing services. One of the most commonly used services is AFS file service, which is protected by Kerberos. Alice, being a very private person, doesn't want others to have access to her files. Through the access control mechanisms provided by AFS, she limits access to specific users. But if these users prefer to access Alice's files through the Web, then the flexibility of AFS access controls disappear.

Web presence for other Kerberized services also suffers. For example, Alice would like to manage her umich.edu X.500 directory entry from a browser. The directory is stored in an LDAP directory that uses Kerberos authentication to control read and write access. Alice would also like to read mail from a browser; this too requires that the Web server authenticates as Alice to the Kerberized mail server.

If an AFS client is running on Alice's workstation, a simple solution presents itself. Instead of making an HTTP request, a user can access AFS file space directly with `file://localhost/afs/`$\cdots$. But it is fair to say that most machines do not run AFS. Also, the solution fails to provide a general mechanism for accessing services from the Web; browsers can not anticipate all possible service access types.

In this scenario, end-to-end authentication presents the question of how to convey Kerberos credentials to the Web server. One solution is for the client to acquire the needed credentials and delegate them to the Web server. A frequently used solution is to send a Kerberos identity and password through SSL, but this gives unlimited power to the Web server to impersonate users, a significant risk. It is also hazardous to expect a user to know when it is safe to give her password to a Web server.

Kerberos supports a mechanism for delegation of rights. However, browsers do not support any form of delegation. A practical solution is needed that works with existing software and is easy to deploy, administer, and maintain. The process should demand minimal interaction with a user, providing transparent access to resources. To limit misuse of user's credentials, the Web server must be constrained in its actions. Furthermore, a central, easily administered location for enforcing security policies controlling the Web server's actions is required.

This paper describes the design, implementation and performance of a system that provides controlled access to Kerberized services through conventional browsers. The system provides a single sign-on through Kerberos authentication: users authenticate once and are given Kerberos and PK credentials. The latter are used for Web authentication. Our system includes a Web server plugin that translates users' PK credentials to Kerberos credentials. Our design assures that Web server actions taken on a user's behalf are limited in time and scope.

The remainder of this paper is organized as follows. Section 2 provides background material and discusses related work. Section 3 presents an architecture for access to Kerberized services through a browser. Section 4 gives implementation details. Section 5 describes performance. Section 6 summarizes and presents directions for future work.

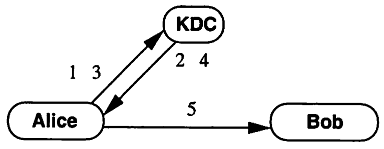
2 Background

First, we review Kerberos, a popular network authentication system based on symmetric key cryptography. Its success stories come from environments with well defined administrative boundaries. We then provide an overview of SSL, a security protocol based on public key cryptography that is universally supported on the Web. The Internet spans many Kerberos realms and requires security solutions that do not have centralized management. SSL provides authenticated and secure connections between any two nodes in the Internet.

We conclude the section with an overview of related work.

2.1 Overview of Kerberos

Kerberos [19] is a network authentication system based on the Needham-Schroeder protocol [17]. Kerberos authentication is illustrated in Figure 1. Authentication is achieved when one party proves knowledge of a shared secret to another. To avoid



LOGIN PHASE:	ONCE PER SESSION
1. *Alice* → *KDC*:	"Hi, I'm *Alice*"
2. *KDC* → *Alice*:	TGT = $\{Alice, \mathrm{TGS}, \mathrm{K}_{A,TGS}\}_{K_{TGS}}$, $\{\mathrm{K}_{A,TGS}, \mathrm{T}\}_{K_A}$
ACCESSING SERVICES:	EVERY TIME BEFORE TALKING TO A SERVICE
3. *Alice* → *TGS*:	*Alice*, *Bob*, TGT, $\{\mathrm{T}\}_{K_{A,TGS}}$
4. *TGS* → *Alice*:	TKT = $\{Alice, Bob, \mathrm{K}_{A,B}\}_{K_B}$, $\{\mathrm{K}_{A,B}, \mathrm{T}\}_{K_{A,TGS}}$
5. *Alice* → *Bob*:	"Hi, I'm *Alice*", TKT, $\{\mathrm{T}\}_{K_{A,B}}$

Figure 1: **Kerberos authentication.** Two phases are shown: initial authentication and service ticket acquisition. KDC is the Kerberos Key Distribution Center. TGS is the Ticket Granting Service. Most implementations combine these services. K_{TGS} is a key shared between the TGS and KDC. K_A is a key shared between *Alice* and the KDC, derived from *Alice's* password. $\mathrm{K}_{A,TGS}$ is a session key for *Alice* and TGS. $\mathrm{K}_{A,B}$ is a session key for *Alice* and *Bob*. T is a timestamp used to prevent replay attacks.

quadratic explosion of key agreement requirements, Kerberos relies on a trusted third party, referred to as a Key Distribution Center (KDC). *Alice*, a Kerberos principal, and *Bob*, a Kerberized service, each establish a shared secret with the KDC.

At login, *Alice* receives a ticket granting ticket, TGT, from the KDC. She uses her password to retrieve a session key encrypted in the reply. The TGT allows *Alice* to obtain tickets from a Ticket Granting Service for other Kerberized services. To access a Kerberized service, *Alice* presents her TGT and receives a service ticket, $\{Alice, Bob, \mathrm{K}_{A,B}\}_{K_B}$. To authenticate to *Bob*, *Alice* constructs a timestamp based authenticator, $\{\mathrm{T}\}_{K_{A,B}}$, proving to *Bob* that she knows the session key inside of the service ticket.

2.2 Overview of SSL/TLS

Secure Socket Layer [1] [12, 13] is a protocol that provides secure connections, addressing the need for entity authentication, confidentiality, and integrity of messages on the Internet. SSL uses public key cryptography, in particular certificates, to accomplish authentication and secret key cryptography to provide confidentiality and integrity of the communication channel. Support for SSL is universal among Web browsers and servers.

[1]SSL is renamed by IETF as Transport Layer Security, TLS [6]

SSL consists of two sub-protocols: the SSL *record* protocol and the SSL *handshake* protocol. The SSL record protocol defines the format used to transmit data. The SSL handshake protocol uses the record protocol to negotiate a security context for a session. SSL supports numerous encryption and digest mechanisms that the client and the server negotiate during the SSL handshake.

Figure 2 shows the exchange of messages in the handshake, details of which are discussed in Section 3.2. Authentication is based on a public key challenge-response protocol [7, 22] and X.509 [11] identity certificates.

SSL supports mutual authentication. First, a user authenticates the server. The user has the responsibility to assure that it can trust the certificate received in the CERTIFICATE message from the server. That responsibility includes verifying the certificate signatures, validity times, and revocation status. The user then sends her public key certificate. The user must also prove that she possesses the private key corresponding to the certificate's public key. For the proof, the user creates a message that contains a digitally signed cryptographic hash of information available to both the user and the server. The server then verifies the signature to be sure that the user possesses the appropriate private key.

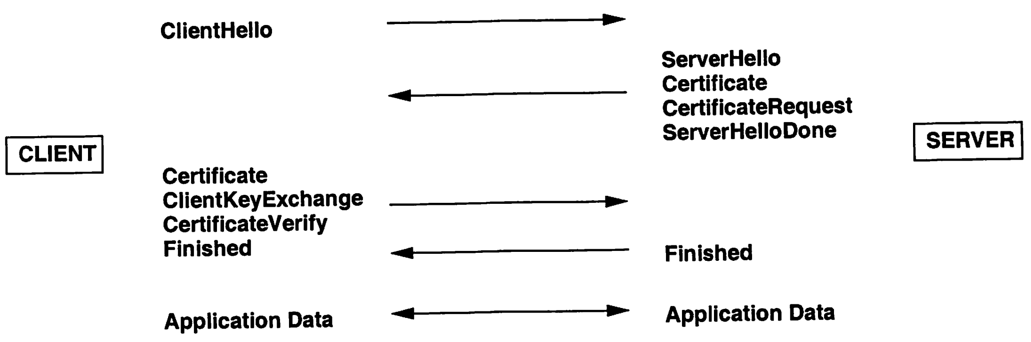


Figure 2: **SSL handshake protocol.** CLIENTHELLO carries a version, random value (part of which is a timestamp), and session id (which allows a user to resume a previous session). A timestamp is used to guarantee the uniqueness of the random value. SERVERHELLO confirms the version and session id. Server sends its CERTIFICATE and requests the user's in CERTIFICATEREQUEST. SERVERHELLODONE specifies the end of a negotiation phase. Client sends her public key certificate in CERTIFICATE. Client sends session information (encrypted with the server's public key) in CLIENTKEYEXCHANGE. Client sends CERTIFICATEVERIFY which contains a signed digest of messages exchanged upto this pointed. The server uses the public key from the client's certificate to verify the client's identity. FINISHED messages serve to end the negotiation process. Secured APPLICATION DATA messages follow. Optional SSL messages are omitted.

To reduce the risk of key compromise, the SSL protocol supports renegotiation of the security context. A client initiates a new handshake by sending a CLIENTHELLO message. If the server wishes to initiate a handshake, it sends an empty SERVERHELLO message to the client and the client responds with a new CLIENTHELLO.

Establishing an SSL session requires sophisticated cryptographic calculations and numerous protocol messages. To minimize the overhead of these calculations and messages, SSL provides a mechanism by which two parties can reuse previously negotiated SSL parameters. With this method, the parties do not repeat the cryptographic operations, they simply resume an earlier session. The user proposes to resume a previous session by including that session's SessionID value in CLIENTHELLO. It is up to the server to decide whether to allow the reuse of the session. We call this a partial SSL handshake.

2.3 Related Work

This section describes related work on distributed authorization and interoperability among authentication mechanisms. Many efforts have focused on creating formal systems that allow reasoning about delegated (restricted) rights and express (general) authorization statements. Many researchers have focused on creating powerful and expressive languages for making and verifying security assertions efficiently. Among them are SPKI/SDSI [5, 10, 21], PolicyMaker, and its successor KeyNote [2, 3, 4], GAA API [23], Akenti [28], and Neuman's proxied authorization [18]. Applications that lack an authorization mechanism of their own greatly benefit from these mechanisms. However, our goal is to make use of already existing authorization mechanisms at the backend services.

There has been work on interoperability of Kerberos and PKI. PKINIT [30] allows a user to use a digital certificate in the initial Kerberos authentication. Public key distributed authentication (PKDA) [24] goes a step further and proposes for Kerberized services to support PK authentication mechanisms. For both PKINIT and PKDA, it is assumed that the user is in direct communication with the server without an interposed Web server.

There is a simple alternative solution to enable the Web server to act on a user's behalf. A user can send his password (securely, of course) to the Web server. The solution has been implemented as an Apache module [1, 27, 25]. In this case, the Web server is given an unlimited power to impersonate users, a significant security risk.

There are several projects that propose to use Kerberos for Web authentication without sending user passwords. Minotaur [9] depends on a client side plugin in to acquire a service ticket for a Web server. However, it has been shown that, in its current design, Minotaur's handling of HTTP POST is insecure. Another system, called SideCar [20], achieves Kerberos authentication by talking to a dedicated process on a client's machine. Failure to start the daemon process prevents the client from being able to do Web authentication. Yet another solution makes use of the extension to TLS cipher suites that includes Kerberos as an authentication mechanism [14].

Kerberos authentication to a Web server is not enough for end-to-end authorization. There must be support for delegating Kerberos credentials after the client authenticates to the Web server, which is addressed by Jackson et al. in their proposal on how to delegate credentials (currently, Kerberos and X509 certificates) in TLS [15]. There are a few problems with considering this approach as a solution. First, no browsers currently support Kerberized TLS. There is an implementation of Kerberized TLS [26] that relies on a local proxy, but browsers are often limited to a single proxy, complicating system management. Furthermore, the description of the exact content of the protocol is vague, making it hard to validate the security of the protocol.

Tuecke et al. [29] propose a specific delegation mechanism that allows a user to delegate an identity certificate to a third party. The receiver must engage in a special verification process that validates these certificates to identify the real sender. Authentication to a commodity server with these certificates cannot be considered secure, as each entity in the delegated path serves as a certification authority and can create a certificate under whatever identity it pleases.

The problem with delegation is that the client may be tricked into requesting a ticket by a rogue server. It has been repeatedly demonstrated that we can not always trust a valid server's certificate, most recently by the Microsoft/VeriSign debacle [16]. Delegation places a large administrative burden on the client. First, a client must be able to understand and apply security policies to determine whether or not to forward his credentials. To avoid the hassle, users frequently allow for unlimited and unchecked delegation. It is not reasonable to assume that for each compromised Web server each user will update her security policy to address the problem. Lastly, browser support for restricted delegation always leaves us wishing for more.

3 Design

Our goal is to design, implement, and deploy a system that allows users access to Kerberized services through a Web server while making use of existing infrastructures and security policies.

The following considerations guide our design.

- The system must use off-the-shelf software whenever possible: conventional Web browsers and servers, Kerberos authentication mechanism, unmodified backend services.
- The solution must not introduce a large burden on system administrators. Administration and management of software is difficult and frequently results in security compromise of the very systems that administrators are trying to protect.
- The solution must not introduce a large burden on the user. The system must be easy to use. Added features should not require user interaction. For example, uses should not be forced to obtain additional credentials.
- The Web server is vulnerable to attacks, so it must be constrained in the actions it is allowed to take on a user's behalf.
- The system must provide a central, easily administered location for policy decisions regarding Web server's actions.

We make the following security assumptions.

- The Web server has adequate physical security.
- The Kerberized Credential Translator, described in Section 3.3, has physical security comparable to the KDC.
- We depend on minimal PKI functionality. We are not trying to solve PKI problems such as reliable and efficient key revocation. This leads to the following additional assumptions.
- We assume the ability to instantiate a root certification authority, be it a self-signed CA certificate or one signed by an acknowledged root CA, such as Versign.

- We assume the CA certificate can be distributed efficiently and securely. All the client machines need to have such a certificate installed in their Web browser CA certificate list (unless the certificate is signed by one of the well acknowledged root CAs). All other servers in the system need to possess the CA certificate.

- We assume the root certificate can be revoked.[2] A mechanism is needed that notifies all clients and servers.

- We assume the (long-lived) certificates issued to the services can be revoked.

Our system consists of components that we describe in detail in the sections below. Section 3.1 describes KX.509, a single sign-on mechanism that produces both Kerberos and PK credentials and creates a binding between them. Section 3.2 discusses client authentication and the Web server's responsibilities in meeting user requests. Section 3.3 introduces our Kerberized Credential Translator, an extension to TGS that converts PK credentials to Kerberos tickets.

3.1 KX.509

In this section, we briefly describe KX.509, a Kerberized service that creates a short-lived X.509 certificate. Doster et al. describe details of the protocol [8].

The exchange of messages and other details of the protocol are shown in Figure 3. As in Kerberos, *Alice* gets a TGT from the KDC. To acquire an X.509 certificate, she first requests a service ticket for a Kerberized Certification Authority, KCA. At the same time, *Alice* generates a public/private key pair and prepares a message for the KCA. Along with the public key, she sends the KCA service ticket, $\{Alice, \mathrm{KCA}, \mathrm{K}_{A,KCA}\}_{K_{KCA}}$, and an authenticator, $\{\mathrm{T}\}_{K_{A,KCA}}$. To ensure that the public key has not been tampered with, the HMAC of the key is sent in the same message. The session key, $K_{A,KCA}$, is used to compute the HMAC of the key.

The KCA authenticates *Alice* by checking the validity of the ticket and the authenticator. It verifies that the public key has not been modified. The KCA then generates an X.509 certificate and sends it back to *Alice*. The certificate is sent in the clear; to prevent tampering, the HMAC of the reply is attached. The lifetime of the certificate is set to the lifetime of the user's Kerberos credentials. The user's Kerberos identity is included inside the certificate, creating the necessary binding.

[2]We know it usually can't.

3.2 Web Server

This section describes the Web server's role in processing a request for a Kerberized service. Our goal is to provide the Web server with a means to access resources on a user's behalf. We built a Web server plugin that engages in proxy authentication by performing the following actions: (i) authenticate the user, (ii) request Kerberos credentials from a credential translator, and (iii) fulfill the user's request by accessing a Kerberized service.

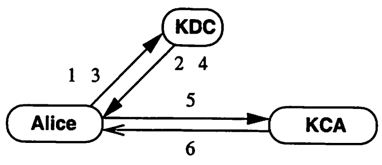
In the first step, client authentication takes place in the SSL handshake. We assume *Alice* possesses a certificate verifiable by the Web server, i.e., the certificate must be issued by a certification authority trusted by the Web server. The client authentication step in SSL requires the user to sign a digest of all the handshake messages prior to CERTIFICATEVERIFY with her private key. SSLv3 uses a keyed digest with the SSL session key. The signature is included in CERTIFICATEVERIFY.

In the second step, the Web server records a transcript of the handshake, details of which are shown in Figure 4. Then, the Web server presents the captured transcript and the SSL session key to a Kerberized Credential Translator (described in Section 3.3) for verification.

In the third step, the Web server uses received credentials to access a Kerberized service. Revealing the SSL session key in the previous step gives the credential translator the power to eavesdrop, so we require the Web server to request renegotiation to establish a new session key, one that is not known to the KCT. This is a trade-off between security and performance. [3]

The user's Kerberos credentials are cached by the Web server to improve performance. The lifetime

[3]One could argue that because the KCT is as powerful as the KDC and can impersonate any user, then the KCT itself can place a request to a Kerberized service, and, thus, the KCT can be trusted with the knowledge of the SSL session key.



1-4	Kerberos login
5. *Alice* → *KCA*:	TKT, Auth, Public Key, $\text{HMAC}_{K_{A,KCA}}$(Public Key)
6. *KCA* → *Alice*:	X.509 certificate, $\text{HMAC}_{K_{A,KCA}}$(X.509 certificate)

Figure 3: **KX.509 protocol.** Steps 1-4 from Kerberos are not shown. Steps 5 and 6 give the details of messages in KX.509. *Alice* sends a service ticket, an authenticator, a public key, and its HMAC. A keyed digest is based on the session key, $K_{A,KCA}$ and prevents modification of the data.

SSL transcript	
1. *Client* → *Server*:	CLIENT HELLO: Version = VC, Random Num = RNC, Session ID = IDC
2. *Server* → *Client*:	SERVER HELLO: Version = VS, Random Num = RNS, Session ID = IDS
3. *Server* → *Client*:	SERVER CERTIFICATE: X.509 certificate = SCert
4. *Server* → *Client*:	SERVER CERTIFICATE REQ: Cert Type = CT, CA chain = CAC
5. *Client* → *Server*:	CLIENT CERTIFICATE: X.509 certificate = CCert
6. *Client* → *Server*:	CLIENT KEY EXCHANGE: $[\text{Key material}]_{K_{WSPK}}$
6. *Client* → *Server*:	CERTIFICATE VERIFY (SSLv3): $[\text{Hash}_{K_{MK}}(\text{VC, RNC, IDC, VS, RNS, CAC, TS, IDS, SCert, CCert})]_{K_{private}}$

Figure 4: **SSL transcript.** The messages listed constitute an SSL transcript. CLIENTHELLO carries a version, random number (first four bytes occupied by a timestamp), and session id, which allows the user to resume a previous session. SERVERHELLO confirms the version and session id. A server sends its CERTIFICATE and requests the user's in CERTIFICATEREQUEST. SERVERHELLODONE specifies the end of the negotiation phase. A client sends her public key certificate in CERTIFICATE. A client sends the session information (encrypted with the server's public key, K_{WSPK}) in CLIENTKEYEXCHANGE. Key material included in this message depends on the key exchange protocol. For example, in the case of RSA, a client generates a premaster secret that both parties use to generate key (encryption and digest) material, including K_{MK}. A client sends CERTIFICATEVERIFY, which includes a key-based digest of all the messages prior to this one signed with the client's private key. The server uses the public key from the client's certificate to verify the client's identity. K_{MK} is the key generated from the key material sent by the client in CLIENTKEYEXCHANGE. We call it the *SSL session key*. $K_{private}$ is the user's private key. A timestamp in CLIENTHELLO is used to verify freshness of the handshake.

of the service ticket issued by the credential translator should be short, minimizing potential misuse of credential. At the same time, the service ticket should have a lifetime long enough that multiple requests from the user do not incur the cost of getting a service ticket each time. A compromise of the Web server enables the intruder to use the currently cached credentials and to acquire credentials on the user's behalf for any of the requests to this compromised Web server.

3.3 Kerberized Credential Translator

We define a *Credential Translator (CT)* as a service that converts one type of credential into another. In this section, we introduce a Kerberized credential translator (KCT) that converts PK credentials to Kerberos credentials.

Figure 5 shows the KCT protocol. First, the Web server authenticates to the KCT by presenting a service ticket, $\{$*Web Server*, KCT, $K_{WS,KCT}\}_{K_{KCT}}$, and the corresponding authenticator, $\{T\}_{K_{WS,KCT}}$. Along with its Kerberos credentials, the Web server sends the SSL transcript, the name of the service ticket being requested, and the SSL session key. After validating the Web server's credentials, the KCT performs the following steps:

- Validates user and server certificates and checks that each was signed by a trusted CA.
- Verifies client's signature in CERTIFICATEVERIFY by recomputing the hash of the handshake messages up to CERTIFICATEVERIFY and comparing it to the corresponding part of the SSL handshake.
- Verifies that the identity inside of the server's certificate matches the Kerberos identity. This step is needed to ensure that the Web server participated in the SSL handshake.
- Assures the freshness of the transcript, by checking the freshness of timestamps or nonces present in the hello messages. In the latter case, the Web server acquires a nonce from the KCT and includes it in SERVERHELLO.
- Generates a service ticket for the user.
- Encrypts the session key included in the service ticket under the Web server's session key, $K_{WS,KCT}$.
- Returns the ticket, authenticator, and encrypted session key to the Web server.
- Logs the transaction for postmortem auditing.

We see that the KCT needs access to the database of keys maintained by the KDC. Consequently, the KCT requires the same physical security as the KDC. In practice, we run the KCT on the same hardware as the KDC, which achieves the physical security requirement and sidesteps the challenge of consistent replication of the Kerberos database.

4 WebAFS Prototype

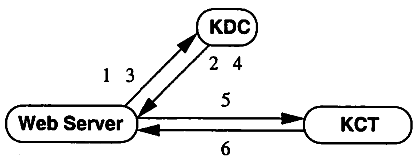
We have implemented a prototype that allows a user to submit requests to a Web server that accesses a Kerberized AFS file server on the user's behalf. An overview of the system is shown in Figure 6. In the remainder of this section, we provide details about the implementation of each of the components involved in the system.

4.1 KX.509

We implemented the KX.509 protocol to work for both Netscape Navigator (on UNIX, Windows, and MacOS) and Internet Explorer (on Windows). The kx509 client and the KCA server are the two basic components involved in issuing users certificates. Additional details about the implementation can be found in a related technical report [8].

Navigator maintains a private cache of certificates, but the implementation is platform dependent, undocumented, and version dependent. Thus, we elect to save certificates in user's Kerberos ticket cache, which requires the user to add a cryptographic module to the browser. No such modification is required for Explorer.

Typically, a ticket cache stores a user's TGT and service tickets. MIT's implementation of Kerberos on UNIX allows for variable size tickets, allowing us to store any data of size up to 1250 bytes, which is sufficient to store a certificate and a private key. Figure



1-4	Original Kerberos done once per lifetime of a session
5. *Web Server* → *KCT*:	TKT, Auth, SSL transcript, $\{\text{MK, Service}\}_{K_{WS,KCT}}$
6. *KCT* → *Web Server*:	TKT=$\{\textit{Alice, Service}, \text{K}_{WS,Service}\}_{K_{Service}}$, $\{\text{K}_{WS,Service}, \text{T}\}_{K_{WS,KCT}}$

Figure 5: **Credential translation protocol.** Steps 1-4, not shown, indicate Kerberos authentication of the Web server. They are performed once per the lifetime of a service ticket for the KCT service. Steps 5 and 6 show the conversation with the KCT. *Service* is the requested backend service. Depending on the version of SSL, an SSL secret key, MK is included in the request to the KCT.

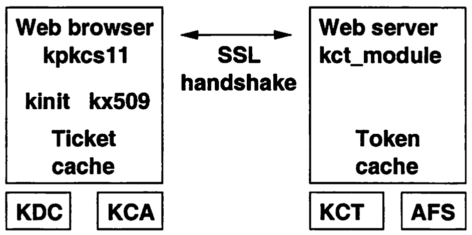


Figure 6: **WebAFS architecture.** We show details of architectural components present in the implementation of the proposed system. The new components are: kpkcs11, kx509, KCA, kct_module, and KCT. The first three components are for credential translation from Kerberos to PK credentials. The last two effect translation in the other direction.

7 shows the output of the `klist` command, which displays the current contents of a ticket cache. The entry `cert.x509/umich.edu@umich.edu` is the service ticket for the KCA. `cert.kx509/umich.edu@umich.edu` contains the user's certificate and private key.

As we mentioned, Navigator needs help to find our certificates. To this end, we use the browser's standard interface to add a cryptographic module that we call `kpkcs11`. When client authentication is required, `kpkcs11` looks up a certificate in the ticket cache and gives it to Navigator.

In our implementation, the user identity information that KCA includes in the certificate is retrieved from a naming service (an X.500 directory). Given a Kerberos principal, KCA looks up the user's first and last name. Additionally, at the end of the distinguished name we attach an email field with the principal name in the local part and the realm in the remote part, for example, `aglo@UMICH.EDU`.

4.2 Web Server

To enable the server to act on a user's behalf, we added a module to the Apache Web server, under 2000 lines of code. The module relies on a version of the `openssl` (0.9.5) library modified to save the SSL transcript. Modifications to the library are minimal (under 200 lines of code) and include a new data structure and calls to a function that saves the incoming and outgoing handshake messages.

We now look more closely at the problems that arise from differences in the SSL protocol specifications and implementations, and from harsh browser realities, which make the solution more complex and introduce delays.

In our prototype, we use timestamps present in SSL handshake to check the freshness of the handshake. Unfortunately, SSLv2 does not include timestamps in the hello messages. Worse yet, Navigator by default starts the SSL handshake with an SSLv2 ClientHello message. Only after receiving the reply from the Web server suggesting the use of SSLv3 does the browser switch to the higher version. The resulting handshake is overall a valid handshake, but lacks an SSLv3 client timestamp. To get the timestamp, we require the Web server to request renegotiation. SSL specifications allow renegotiation only after the ongoing handshake is complete, so two full SSL handshakes must take place.

One feature of the SSL protocol, called a partial handshake, requires special attention. When a partial SSL handshake happens, the Web server checks if AFS credentials are cached; if so, then the server proceeds with the AFS request. Otherwise, the Web server forces an SSL renegotiation followed by a full SSL handshake. After creating a transcript, the Web server, as before, submits a request to the KCT for an AFS service ticket.

4.3 Kerberized Credential Translator

The responsibilities of the KCT are to verify the validity of the request and issue an AFS ticket on the user's behalf. To fulfill this role the KCT must have special privileges: it must be able to read the KDC database and use the key of the AFS Kerberos principal. Currently, tickets are issued only for AFS. In deployment, the Web server will specify the service for which it needs a ticket, at which point the KCT will need a security policy to make authorization decisions about who can ask for what.

As of this writing, the MIT Kerberos libraries are not thread-safe, so the KCT cannot be implemented as a multithreaded application. To improve performance, we spawn a process to handle incoming requests. To achieve the required physical security, we run the KCT on the same hardware as the KDC. Implementation of the KCT is under 2000 lines of code.

5 Performance

In this section we discuss the performance of the system by examining the cost of making a request to a Web server, which, in turn, requests a service from a backend server on a user's behalf. The experiments described in this section were performed on an unloaded Intel 133MHz Pentium workstation running RedHat Linux 6.2 (kernel version 2.2). Our focus is on understanding overhead induced by the system, so all the components were executed on the same hardware to avoid network and file access delays.

The software was tested against commodity

```
$:  klist
Kerberos 4 ticket cache:  /tmp/tkt500
Principal:  aglo@UMICH.EDU

Issued              Expires             Principal
01/19/01 13:39:56   01/19/01 23:39:56   krbtgt.UMICH.EDU@UMICH.EDU
01/19/01 13:40:07   01/19/01 23:39:56   cert.x509@UMICH.EDU
01/19/01 13:40:07   01/19/01 23:39:56   cert.kx509@UMICH.EDU
```

Figure 7: **Output of klist.** KX.509 certificate and the private key are stored in the Kerberos IV ticket cache under the service names of `cert.kx509`. `cert.x509` is the service ticket for the KCA. the other entry is the service ticket for the TGS.

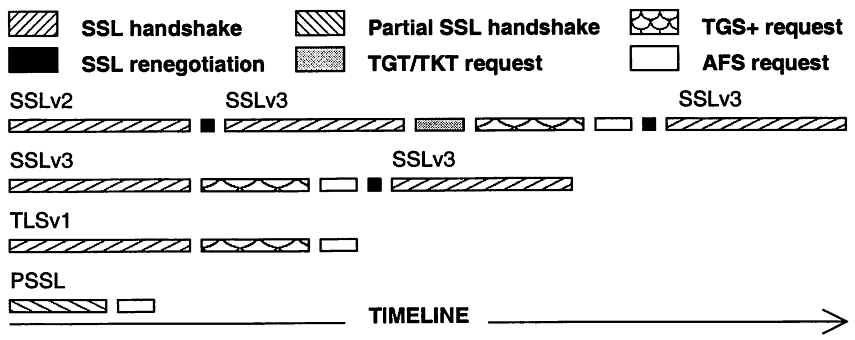


Figure 8: **Timelines for WebAFS requests.** We show the components of a user request in four scenarios illustrated as timelines. The legend identifies each of the components involved. We consider all the different versions of an SSL protocol, v2, v3, TLSv1, and a partial handshake. Access to an AFS file server is used as an example.

browsers, but it is hard to glean detailed measurements from commercial software, so we used `openssl` tools to mimic the browser's actions. We used `openssl`'s generic SSL/TLS client, called `s_client`. All requests were made for a 1K file. For each of the test cases 30 trials were measured and averaged.

We define a *browser session* to be the time from launch to termination of the browser application. We define a *server session* to be the time from the first request to a Web server until the termination of the browser application. Within a browser session a user starts multiple server sessions. Requests for different files from the same Web server fall into a single server session. Requests to different Web servers are associated with different server sessions.

Figure 8 shows the breakdown of a user's request into the basic components. Four scenarios are illustrated as timelines. A typical request consists of some of the following stages:

1. SSLv2 handshake (or a partial SSL handshake)
2. Request renegotiation
3. SSLv3 handshake
4. Refresh Web server's Kerberos credentials
5. Request Kerberos credentials from KCT
6. Request renegotiation
7. SSLv3 handshake

Components	Delay(s)
1 handshake	1.25
2 handshakes	2.50
TGT/KCT_TKT	0.03
KCT request	0.26
Partial SSL	0.02

Table 1: **Delays of basic components.** The first row shows SSL handshake latency. The second row shows the delays seen after two consecutive SSL handshakes with a request to renegotiate in between. The third row shows the time for the Web server to refresh Kerberos credentials. The fourth row shows delays associated with the KCT request/reply. The last row shows the latency for a partial SSL handshake. Rows 1, 2, and 5 reflect end-to-end delays seen by the user. Rows 3 and 4 measure latencies seen by the Web server while talking to the KDC and KCT.

We divide a user's request into different components, for example an SSL handshake, and measured each of the components individually. Table 1 shows the delays associated with the basic components involved in a user's request.

End-to-End	Time(s)
SSLv2 hello no TGT	4.08
SSLv2 hello 1st request	4.04
SSLv2 cached creds	2.50
SSLv3 hello no TGT	2.86
SSLv3 hello request	2.80
SSLv3 cached creds	1.25

Table 2: **End-to-end delays.** Each of the scenarios represents a possible user request. We measured end-to-end latency seen by the user.

Table 2 shows the end-to-end delays seen by the user for different types of requests. We describe each of the scenarios in detail and point out which ones are more common. We divide requests into two groups, depending on whether user's credentials are cached at the Web server.

- **No cached credentials.** First, we consider the cases where user's credentials are not cached. This happens when a user is making the first request to the Web server or when her credentials have been evicted from the Web server's LRU cache.
    - **Once a day:** *SSLv2 hello no TGT* and *SSLv3 hello no TGT*. In these two scenarios, the Web server has stale credentials so the user's request gets penalized by the time needed by the Web server to get new Kerberos credentials. The lifetime of our Web server's TGT is 24 hours.
    - **Once per server session:** *SSLv2 hello 1st request*. When contacting a Web server for the first time, the default behavior of Navigator is to start with an SSLv2 ClientHello message. Until the browser is restarted, all subsequent requests will start with an SSLv3 ClientHello. This scenario measures the overhead of the three handshakes and a KCT request. The first additional handshake produces a valid timestamp in the ClientHello message. The second additional handshake renegotiates the SSL session key, which was revealed to the KCT.
    - **Most common request:** *SSLv3 hello request*. Explorer starts with an SSLv3 ClientHello. Any requests from this browser fall either into this category or the partial handshake.
- **Cached credentials.** We now review the scenarios where the user's credentials are cached at the Web server. Caching is important because it saves

the overhead of getting Kerberos credentials. Furthermore, no SSL renegotiation plus handshake is needed at the end. The only overhead the system imposes is that associated with token management.

- **Frequent:** *Partial handshake cached credentials.* The lifetime of the session key negotiated in the full handshake is configurable by the web server. If more than one request is made within five minutes of a full handshake, a partial handshake takes place. (Five minutes is a default value used by Apache Web servers). We can safely assume that user's credentials are already cached at that point. The time required for a partial handshake is considerably smaller than for a full handshake. The frequency of these requests depends on the user's access pattern.

- **Common:** *SSLv3/TLSv1 cached credentials.* Once the user contacts a Web server, her credentials are cached until they get evicted due to expired lifetime or lack of space. When requests to the Web server are separated by more than five minutes, a user experiences end-to-end delay presented in last row of Table 2.

- **Unlikely:** *SSLv2 hello cached credentials.* The browser sends an SSLv2 ClientHello message to the Web server if it never contacted it within the current browser session. However, it is still possible for the user's credentials to be cached at the Web server if the user restarted the browser within the lifetime of the cached credentials.

To summarize, an SSL handshake costs 1.25 seconds. Delays associated with refreshing a TGT and making KCT requests are small: 0.02 and 0.26 seconds, respectively.

In the most common case, credentials are cached and SSLv3 connections are used, so the system incurs negligible overhead. Further testing in more complex environments is necessary and will be done in the future. However, these preliminary results are encouraging.

6 Discussion

In this paper we described a system that provides users with access to Kerberized services through a browser. In this section we summarize the functionality of each of the components involved in the system and point out the issues that require further research.

While many backend services use Kerberos for authentication, Web servers use SSL to authenticate with public key cryptography. We address the mismatch of authentication credentials between the Web server and Kerberized service by introducing a new service that translates PK credentials to Kerberos tickets. The Web server engages in proxy authentication. The process consists of SSL client authentication, a request to a credential translation service, and finally authentication to the Kerberized service on a user's behalf.

We built a single sign-on mechanism that allows users to obtain X.509 certificates in addition to their Kerberos credentials. Through the KX.509 protocol, we create a binding between a user's Kerberos and PK identities. The issues surrounding this binding are quite broad and require further study.

A client uses her certificate to establish an authenticated and secured channel to a Web server. The Web server logs the SSL transcript and makes an authenticated request to a new service that translates the user's PK credentials to Kerberos credentials.

The authorization model of the credential translator is primitive and is the focus of our future work. The current model supports generic access control lists: for each Web server there is an entry listing the Kerberized services for which it can request tickets. We are looking into integrating Akenti [28] access control mechanisms into the system.

We built a prototype, WebAFS, that allows users to access restricted AFS files through browsers. It requires minor modifications to existing software, such as a plugin module to Navigator and modifications to the `openssl` library. We wrote four components: `kx509` and KCA take care of issuing user's certificate, an Apache module services requests, and KCT translates between two types of credentials.

We measured the overhead introduced by our system. We showed the delays associated with the building blocks of a user's request. The results show that a substantial amount of time is spent in establishing an SSL connection, but that requesting credentials for the server is nicely amortized over a browser session.

Credential translation need not apply only to Web traffic. It is extensible to any SSL-enabled client and SSL-enabled server communication. Furthermore, credential translation need not be limited to producing Kerberos credentials. Consider a remote login application such as an SSL-enabled Telnet. Assuming a user has a certificate on his local computer, we can obviate the need to send his password over the network. A user can use his certificate, mutually authenticate with the remote host, and empower it to act on his behalf. We are considering these and other extensions in our future work.

7 Acknowledgments

We thank the anonymous reviewers for their helpful comments. We also thank CITI staff for their participation in the project and their valuable comments. Our special thanks go to Dr. Naomaru Itoi for many insights and constant encouragement.

References

[1] Apache Software Foundation. Apache web server. `http://www.apache.org`.

[2] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. The KeyNote trust managment system version 2, September 1999. RFC 2704.

[3] M. Blaze, J. Feigenbaum, and A. Keromytis. Keynote: Trust management for public-key infrastructure. In *Proceedings Cambridge 1998 Security Protocols International Workshop*, 1998.

[4] M. Blaze, J. Feigenbaum, and M. Strauss. Compliance checking in the PolicyMaker trust management system. In *Proceedings of Second International Conference on Financial Cryptography*, 1998.

[5] D. Clarke, J. Elien, C. Ellison, F. Morcos, and R. Rivest. Certificate chain discovery in SPKI/SDSI. Draft Paper, November 1999.

[6] T. Dierks and C. Allen. The TLS protocol version 1.0, January 1999. RFC 2246.

[7] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transaction on Information Theory*, 22:644–654, November 1976.

[8] W. Doster, M. Watts, and D. Hyde. The KX.509 protocol. CITI Technical Report 01-2, February 2001.

[9] P. Dousti. Project Minotaur: Kerberizing the Web, software at Carnegie Mellon University. `http://andrew2.andrew.cmu.edu/minotaur`.

[10] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI certificate theory, September 1999. RFC 2693.

[11] ITU-T (formerly CCITT) Information technology Open Systems Interconnection. Recommendation X.509: The directory authentication framework, December 1988.

[12] A. Freier, P. Karton, and P. Kocher. Secure Socket Layer 3.0, March 1996. Internet draft.

[13] A. Freier, P. Karton, and P. Kocher. The SSL protocol version 3.0, March 1996. Netscape Communications Corporation.

[14] M. Hur and A. Medvinsky. Kerberos cipher suites in Transport Layer Security (TLS), May 2001. Internet draft.

[15] K. Jackson, S. Tuecke, and D. Engert. TLS delegation protocol, February 2001. Internet draft.

[16] Microsoft Security Bulletin MS01-017. Erroneous VeriSign-issued digital certificates pose spoofing hazard, March 2001.

[17] R. Needham and M. Shroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993 – 999, December 1978.

[18] C. Neuman. Proxy-based authorization and accounting for distributed systems. In *Proceedings of the 13th International Conference on Distributing Computing Systems*, May 1993.

[19] C. Neuman and T. Ts'o. Kerberos: an authentication service for computer networks. *IEEE Communications*, 32(9):33–38, September 1994.

[20] SideCar project. Software at Cornell University. `http://www.cit.cornell.edu/kerberos/sidecar.html`.

[21] R. Rivest and B. Lampson. SDSI – A simple distributed security infrastructure. Presented at CRYPTO'96 Rump session, 1996.

[22] R. Rivest, A. Shamir, and L. Adleman. A method of obtaining digital signatures and public key cryptosystems. *Communications of the ACM*, 21:120–126, February 1978.

[23] T. Ryutov and C. Neuman. Representation and evaluation of security policies for distributed system services. In *Proceedings of the DISCEX*, January 2000.

[24] M. Sirbu and J. Chuang. Distributed authentication in Kerberos using public key cryptography. In *Symposium On Network and Distributed System Security*, 1997.

[25] D. Song. Kerberized WWW access. `http://www.monkey.org/~dugsong/krb-www`.

[26] V. Staats. Kerberized TLS, June 2000. Private communication.

[27] Stone Cold Software. Apache Kerberos Module. `http://stonecold.unity.ncsu.edu`.

[28] M. Thompson, W. Johnson, S. Mudumbai, G. Hoo, K. Jackson, and A. Essiari. Certificate based access control for widely distributed resources. In *Proceedings of the 8th USENIX Security Symposium*, August 1999.

[29] S. Tueke, D. Engert, and M. Thompson. Internet X.509 public key infrastructure impersonation certificate profile, February 2001. Internet draft.

[30] B. Tung, C. Neuman, and J. Wray. Public key cryptography for initial authentication in Kerberos, April 2000. Internet draft.

Dos and Don'ts of Client Authentication on the Web

Kevin Fu, Emil Sit, Kendra Smith, Nick Feamster
{fubob, sit, kendras, feamster}@mit.edu
MIT Laboratory for Computer Science
http://cookies.lcs.mit.edu/

Abstract

Client authentication has been a continuous source of problems on the Web. Although many well-studied techniques exist for authentication, Web sites continue to use extremely weak authentication schemes, especially in non-enterprise environments such as store fronts. These weaknesses often result from careless use of authenticators within Web cookies. Of the twenty-seven sites we investigated, we weakened the client authentication on two systems, gained unauthorized access on eight, and extracted the secret key used to mint authenticators from one.

We provide a description of the limitations, requirements, and security models specific to Web client authentication. This includes the introduction of the *interrogative adversary*, a surprisingly powerful adversary that can adaptively query a Web site.

We propose a set of hints for designing a secure client authentication scheme. Using these hints, we present the design and analysis of a simple authentication scheme secure against forgeries by the interrogative adversary. In conjunction with SSL, our scheme is secure against forgeries by the active adversary.

1 Introduction

Client authentication is a common requirement for modern Web sites as more and more personalized and access-controlled services move online. Unfortunately, many sites use authentication schemes that are extremely weak and vulnerable to attack. These problems are most often due to careless use of authenticators stored on the client. We observed this in an informal survey of authentication mechanisms used by various popular Web sites. Of the twenty-seven sites we investigated, we weakened the client authentication of two systems, gained unauthorized access on eight, and extracted the secret key used to mint authenticators from one.

This research was supported by a USENIX scholars fellowship.

This is perhaps surprising given the existing client authentication mechanisms within HTTP [16] and SSL/TLS [11], two well-studied mechanisms for providing authentication secure against a range of adversaries. However, there are many reasons that these mechanisms are not suitable for use on the Web at large. Lack of a central infrastructure such as a public-key infrastructure or a uniform Kerberos [41] contributes to the proliferation of weak schemes. We also found that many Web sites would design their own authentication mechanism to provide a better user experience. Unfortunately, designers and implementers often do not have a background in security and, as a result, do not have a good understanding of the tools at their disposal. Because of this lack of control over user interfaces and unavailability of a client authentication infrastructure, Web sites continue to reinvent weak home-brew client authentication schemes.

Our goal is to provide designers and implementers with a clear framework within which to think about and build secure Web client authentication systems. A key contribution of this paper is to realize that the Web is particularly vulnerable to adaptive chosen message attacks. We call an adversary capable of performing these attacks an *interrogative adversary*. It turns out that for most systems, every user is potentially an interrogative adversary. Despite having no special access to the network (in comparison to the eavesdropping and active adversary), an interrogative adversary is able to significantly compromise systems by adaptively querying a Web server. We believe that, at a minimum, Web client authentication systems should defend against this adversary. However, with this minimum security, sites may continue to be vulnerable to attacks such as eavesdropping, server impersonation, and stream tampering. Currently, the best defense against such attacks is to use SSL with some form of client authentication; see Rescorla [37] for more information on the security and proper uses of SSL.

In Section 2, we describe the limitations, requirements, and security models to consider in designing Web client authentication. Using these descriptions, we codify the principles underlying the strengths and weaknesses of existing systems as a set of hints in Section 3. As an example, we design a simple and flexible authentication scheme in Section 4. We implemented this scheme and analyzed its security and performance; we present these findings in Sections 5 and 6. Section 7 compares the work in this paper to prior and related work. We conclude with a summary of our results in Section 8.

2 Security models and definitions

Clients want to ensure that only authorized people can access and modify personal information that they share with Web sites. Similarly, Web sites want to ensure that only authorized users have access to the services and content it provides. Client authentication addresses the needs of both parties.

Client authentication involves proving the identity of a *client* (or user) to a *server* on the Web. We will use the term "authentication" to refer to this problem. Server authentication, the task of authenticating the server to the client, is also important but is not the focus this paper.

In this section, we present an overview of the security models and definitions relevant in client authentication. We begin by describing the practical limitations of a Web authentication system. This is followed by a discussion of common requirements. We then characterize types of breaks and adversaries.

2.1 Practical limitations

Web authentication is primarily a practical problem of deployability, user acceptability, and performance.

Deployability

Web authentication protocols differ from traditional authentication protocols in part because of the limited interface offered by the Web. The goal is to develop an authentication system by using the protocols and technologies commonly available in today's Web browsers and servers. Authentication schemes for the Internet at large cannot rely on technology not widely deployed. For example, Internet kiosks today do not have smart card readers. Similarly, home consumers currently have little incentive to purchase smart card readers or other hardware token systems.

The client generally speaks to the server using the Hypertext Transfer Protocol (HTTP [14]). This may be spoken over any transport mechanism but is typically either TCP or SSL. Since HTTP is a stateless, sessionless protocol, the client must provide an *authentication token* or *authenticator* with each request.

Computation allows the browser to transform inputs before sending them to the server. This computation may be in a strictly defined manner, such as in HTTP Digest authentication [16] and SSL, or it may be much more flexible. Flexible computation is available via Javascript, Java, Tcl, ActiveX, Flash, and Shockwave. Depending on the application, these technologies could perhaps assist in the authentication process. However, most of these technologies have high startup overhead and mediocre performance. As a result, users may choose to disable these technologies. Also, these extensions may not be available on all operating systems and architectures. Any standard authentication scheme should be as portable and lightweight as possible, and therefore require few or no browser extensions. Thus for today's use, any authentication scheme should avoid using client computation for deployability reasons. If absolutely necessary, Javascript and Java are commonly supported.

Client state allows the client's browser to store and reuse authenticators. However, storage space may be very limited. In the most limited case, the browser can only store passwords associated with realms (as in HTTP Basic authentication [16]). A more flexible form of storage which is commonly available in browsers is the cookie [25, 32]. Cookies allow a server store a value on a client. In subsequent HTTP requests, the client automatically includes the cookie value. A number of attributes can control how long cookies are kept and to which servers they are sent. In particular, the server may request that the client discard the cookie immediately or keep it until a specified time. The server may also request that the client only return the cookie to certain hosts, domains, ports, URLs, or only over secure transports. Cookies are the most widely deployed mechanism for maintaining client state.

User acceptability

Web sites must also consider user acceptability. Because sites want to attract many users, the client authentication must be as non-confrontational as possible. Users will be discouraged by schemes requiring work such as installing a plug-in or clicking away dialog boxes.

Performance

Stronger security protocols generally cost more in performance. Service providers naturally want to respond to as many requests as possible. Cryptographic solutions will usually degrade server performance. Authentication should not needlessly consume valuable server resources such as memory and clock cycles. With current technology, SSL becomes unattractive because of the computational cost of its initial handshaking.

2.2 Server security requirements

The goals of a server's authentication system depend on the strength and granularity of authentication desired. Granularity refers to the fact that some servers identify individual users throughout a session, while others identify users only during the first request. A fine-grained system is useful if specific authorization or accountability of a user is required. A coarse-grained system may be preferred in situations where partial user anonymity is desired.

A simple example of a coarse-grained service is a *subscription service* [42]. Subscription services merely wish to verify that a user has paid for the service before allowing access to read-only content. During the initial request, a user could authenticate with a username and password. Unless the service allows customization, subsequent requests need only verify that a user has been authenticated without knowing the user's actual identity. A trusted third party could handle the initial authentication of a user. Some specific examples of sites that only require this level of authentication are newspapers (e.g., `WSJ.com`), online libraries (e.g., `acm.org`), and adult entertainment (e.g., `playboy.com`).

However, most sites customize the data sent back to users. This naturally requires a fine-grained system. Each user must be identified specifically to use their preferences. Examples of this include sites that allow users to customize look-and-feel (e.g., `slashdot.org`), sites that filter information on behalf of the user (e.g., `infobeat.com`), or sites which provide online identities (e.g., `hotmail.com`).

2.3 Confidentiality and privacy

Confidentiality is not strictly related to authentication but it is worth mentioning as well, since it can be provided by cryptography and since it is often confused with authentication. A system that provides confidentiality protects traffic from disclosure by anyone except the sender and recipient. In contrast, a system that provides authentication ensures that the person sending or receiving the data is indeed who they claim to be. This may be confusing because SSL, the only widely deployed mechanism for providing confidentiality of HTTP transactions, provides options for both authentication and confidentiality. The distinction between confidentiality and authentication is further blurred by the practice of current browsers of displaying a single padlock whose meaning is ambiguous.

Typically, servers choose to provide confidentiality for only certain special data by using SSL. For example, financial data require confidentiality. Sites that deal with such information, *online brokerages*, may be auction sites (e.g., `ebay.com`), banks and other financial service providers (e.g., `etrade.com`), or online merchants (e.g., `FatBrain.com`).

Another issue commonly associated with authentication is user *privacy*. Privacy refers to protecting the data available on the server from access by unauthorized parties. While often the information provided by the server is not itself secret, one does not usually want unknown parties discovering their personal interests. For example, a user may sign up to see discount airfares to San Francisco or select stocks in a portfolio for updated stock quotes. While the fact that US Airways is offering a low fare or that Cisco stock has shed four points is not in any way secret, it may be telling to find out if a particular user is interested in that information. Therefore servers often need to provide ways to keep personalized data private. Privacy can be achieved by using secure authentication and providing confidentiality.

2.4 Breaks

An adversary's goal is to break an authentication scheme faster than by brute force. Here we use terminology loosely borrowed from cryptography to characterize the kinds of breaks an adversary can achieve [19, 31].

In an *existential forgery*, the adversary can forge an authenticator for at least one user. However, the adversary cannot choose the user. This may be most interesting in the case where authenticators protect access to a subscription service. While an existential forgery would not give an adversary access to a chosen user's account, it would allow the adversary to access content without paying for it. This is the least harmful kind of forgery.

In a *selective forgery*, the adversary can forge an authenticator for a particular user. This adversary can access any chosen user's personalized content, be it Web e-mail or bank statements.

Note that a forgery implies the construction of a new authenticator, not one previously seen. In a traditional *replay attack*, the adversary is merely reusing a captured authenticator.

Finally, a *total break* results in recovery of the secret key used to mint authenticators. This is the most serious break in that it allows the adversary to construct valid authenticators at any time for all users.

2.5 Adversaries

We consider three kinds of adversaries that attack Web authentication protocols: the *interrogative adversary*, the *eavesdropping adversary*, and the *active adversary*. Each successive adversary possesses all the abilities of the weaker adversaries. Note that our definitions differ somewhat from tradition. Our adversaries gather information and apply this information to achieve a break. The adversaries differ from each other only in their information gathering ability.

Interrogative adversary

The *interrogative adversary* can make a reasonable number of queries of a Web server. It can adaptively choose its next query based on the answer to a previous query. We named this the interrogative adversary because the adversary makes many queries, but lacks the ability to sniff the network.

The ability to make queries is surprisingly powerful. The adversary can pass attempted forgeries to the server's verification routine. By creating new accounts on a server, the adversary can obtain the authenticator for many different usernames. This is possible on any server that allows account creation without some form of out-of-band authentication (e.g., credit cards) to throttle requests. In this paper we assume no such throttle exists.

The interrogative adversary can also use information publicly available on the server. A server may publish the usernames of valid account holders, perhaps in a public discussion forum. An adversary attacking this server might find this list useful.

In more theoretical terms, the interrogative adversary may treat the server as an oracle. An interrogative adversary can carry out an *adaptive chosen message attack* by repeatedly asking for the server to mint or verify authenticators [19].

Eavesdropping adversary

The *eavesdropping adversary* can see all traffic between users and the server, but cannot modify any packets flowing across the network. That is, the adversary can sniff the network and replay authenticators. This adversary also has all the abilities of the interrogative adversary.

An eavesdropping adversary can apply its sniffed information to attempt a break. Computer systems research would consider this an active attack; we do not. This style of definition is more common in the theory community where attacks consist of an information gathering process, a challenge, another optional information gathering process, and then an attempted break [3].

Active adversary

The *active adversary* can in addition see and modify all traffic between the user and the server. This adversary can mount man-in-the-middle attacks. In the real world, this situation might arise if the adversary controls a proxy service between the user and server.

3 Hints for Web client authentication

We present several hints for designing, implementing, and selecting a scheme for client authentication on the Web. Some of these hints come from our experiences in breaking authentication schemes in use on commercial Web sites. Others come from general knowledge or security discussion forums [46]. Following these hints is neither necessary nor sufficient for security. However, they would have prevented us from breaking the authentication schemes on several Web sites mentioned in this section. Most of these sites have subsequently repaired the problems we identified. These incidents help to demonstrate the usefulness of these hints. The details of our analysis are documented in our technical report [18].

Although we give advice on how to perform client authentication on the Web, we certainly do not advocate having everyone design their own security systems. Rather, we hope that these hints will assist researchers and developers of Web client authentication and dissuade persons unfamiliar with security from implementing home-brew solutions.

The hints fall into three categories. Section 3.1 discusses the appropriate use of cryptography. Section 3.2 explains why passwords must be protected. Section 3.3 offers suggestions on how to protect authenticators.

3.1 Use cryptography appropriately

Use of cryptography is critical to providing authentication. Without the use of cryptography, it is not possible to protect a system from the weakest of adversaries. However, designing cryptographic systems is a difficult and subtle task. We offer some hints to help guide the prospective designer in using the cryptographic tools available.

Use the appropriate amount of security

An important general design hint is to Keep It Simple, Stupid [27]. The more complex the scheme, the harder it is to develop compelling arguments that it is secure. If you are designing or selecting a system, choose one that provides the right amount of security for your needs. For example, an online newspaper cares about receiving compensation for content. An online brokerage cares about confidentiality, integrity, and authentication of information. These security needs are very different and can be satisfied by different systems. There are usually tradeoffs between the user interface, usability, and performance. Choosing an overly complex or featureful system will make management more difficult; this can easily result in security breaches.

Do not be inventive

It is a general rule in cryptography that secure systems should be designed by people with experience. Time has repeatedly shown that systems designed or implemented by amateurs are weak and easily broken. Thus, while we encourage research in developing authentication systems for the Web, it is very risky to design your own authentication system. This is closely related to our next hint. If you do choose to implement your own scheme, you should make your protocol publicly available for review.

Do not rely on the secrecy of a protocol

A security system should not rely on the secrecy of its protocol. A protocol whose security relies on obscurity is vulnerable to an exposure of the protocol. If there are any flaws, such an exposure may reveal them. For example, a secret system can be probed by an interrogative adversary to determine its behavior to valid and invalid inputs. This technique allowed us to reverse engineer the `WSJ.com` client authentication protocol. By creating several valid accounts and comparing the authenticators returned by the system, we were able to determine that the authenticator was the output of `crypt` (salt, username + secret string) where + denotes concatenation. Once we understood the format of the authenticator, we were able to quickly recover the secret string, "March20", by mounting an adaptive chosen message attack. The program, included in the technical report [18], runs in 128×8 queries rather than the intended 128^8. Assuming each query takes 1 second, this program finishes in 17 minutes instead of the intended 2×10^9 years. This information constitutes a total break, allowing us to mint valid authenticators for all users.

On the other hand, Open Market published their design and implementation [29], and Yahoo [47] provided us with the complete details of their authentication system. We believe these schemes are reasonably strong; for more details see the relevant sections of our technical report [18].

Instead of relying on the secrecy of the scheme, rely on the secrecy of a well-selected set of keys. Ensure that the protocol is public so that it can be reviewed for flaws and improved. This will lead to a more secure system than a private protocol which appears undefeatable but may in practice be fairly easy to break. If you are hesitant to reveal the details of an authentication scheme, then it may be vulnerable to attack by an interrogative adversary.

Understand the properties of cryptographic tools

When designing an authentication scheme, cryptographic tools are critical. These include hash functions such as SHA-1 [15], authentication codes like HMAC [24], and higher-level protocols like SSL [11]. The properties each tool must be understood.

For example, SSL alone does not provide user authentication. Although SSL can authenticate users with X.509 client certificates, commercial Web sites rarely use this feature because of PKI deployment problems. Instead, SSL is used to provide confidentiality for authentication tokens and data. However, confidentiality does not ensure authentication.

Misunderstanding the properties of SSL made `FatBrain.com` vulnerable to selective forgeries by an interrogative adversary. In an earlier scheme, their authenticator consisted of a username and a session identifier based on a global sequence number. Since this number was global, an interrogative adversary could guess the session identifier for a chosen victim and make an SSL request with this session identifier. Here, the use of SSL did not make the system secure.

A more detailed example comes from a misuse of a hash function. One commonly (and often incorrectly) used input-truncating hash function is the Unix `crypt()` function. It takes a string input and a two-character salt to create a thirteen-character hash [31]; it is believed to be almost as strong as the underlying cryptographic cipher, DES [44]. However, `crypt()` only considers the first eight characters of its string input. This truncation property must be taken into account when using it as a hash.

The original `WSJ.com` authentication system failed to do so, which made our break possible. Since the input to `crypt()` was the username concatenated with the server secret, the truncation property of `crypt()` meant that the secret would not be hashed if the username was at least eight characters long. This means authenticators for long usernames can be easily created, merely with knowledge of the username. Additionally, the algorithm will produce an identical authenticator for all usernames that match in the first eight characters. This can be seen in Figure 1.

It is likely that `WSJ.com` expected this construction to act like a secure message authentication code (MAC). A message authentication code is a one-way function of both its input and a secret key that can be used to verify the integrity of the data [43]. The output of the function is deterministic and relatively short (usually sixteen to twenty bytes). This means that it can be recalculated to verify that the data has not been tampered with.

However, the `WSJ.com` authenticator was just a deterministic value which could always be computed from the first eight characters of the username and a fixed secret. While HTTP Basic authentication [16] (which uses no cryptography at all) is secure against an existential forgery of an interrogative adversary, the original `WSJ.com` scheme fell to a total break by the interrogative adversary.

Thus, when possible you should use a secure message authentication code. Certain cryptographic constructions have subtle weaknesses [31], so you should take great care in choosing which algorithm to employ. We recommend the use of HMAC-SHA1 [24]. This algorithm prevents many attacks known to defeat simple constructions. However, as we will see in Section 6, use of secure message authentication code is more expensive than an input-truncating hash such as `crypt()`.

Do not compose security schemes

It is difficult to determine the effects of composing two different security systems. Breaking one may allow an adversary to break the other. Worse, simply composing the schemes may have adverse cryptographic side effects, even if the schemes are secure in isolation. Menezes et al explain in remark 10.40 how using a single key pair for multiple purposes can compromise security [31]. The use of a single key for authentication and confidentiality leads to compromise of both if that key is stolen. On the other hand, if separate keys are used, a break of the authentication will not affect the confidentiality of past messages and vice versa.

`FatBrain.com` had two separate user authentication systems. To purchase a book, a user entered a username and password at the time of purchase. Future purchases required reauthentication. The account management Web pages had a separate security scheme which was stateful. After the user entered a username and password, FatBrain established a session identifier in the URL path. In this way, users could navigate to other parts of the account management system without having to tediously re-enter the password. Unfortunately, the security hole discussed in Section 3.3 allowed an adversary to gain access to the account management system for an arbitrary user by guessing a valid session identifier. The account management system includes an option to change a user's registered email address. By changing the email address of a victim's account and then selecting "mail me my password," an adversary could break into to the book purchasing part of the system, despite the fact that it was secure in isolation.

3.2 Protect passwords

Passwords are the primary means of authenticating users on the Web today. It is important that any Web site guard the passwords of its users carefully. This is especially important since users, when faced with many Web sites requiring passwords, tend to reuse passwords across sites.

Limit exposure of passwords

Compromise of a password completely compromises a user. A site should never reveal a password to a user. For instance, `ihateshopping.net` included the user's password as a hidden form variable. A valid user should already know the password; sending it unnecessarily over the network gives the eavesdropping adversary more opportunity to sniff the password. Furthermore, sites should use the "password" field type in HTML forms. This hides the password as it is typed in and prevents an adversary from peeking over a user's shoulder to copy the password.

Even for non-secure Web sites, users should have the option to authenticate over SSL. That is, users should not type passwords over HTTP. Passwords sent over HTTP are visible to eavesdropping adversaries sniffing the network and active adversaries impersonating servers. Because users often have the same password on multiple servers, a stolen password can be extremely damaging. To protect against such attacks, a server could require users to conduct the login over an SSL connection to provide confidentiality for the password exchange; upon successful completion of the login exchange, the server can then set a cookie with an unforgeable authenticator for use over HTTP. The authenticator can be designed to limit the spread of damage, whereas passwords can not.

username	`crypt()` output	authentication cookie
bitdiddle	MaRdw2J1h6Lfc	bitdiddleMaRdw2J1h6Lfc
bitdiddler	MaRdw2J1h6Lfc	bitdiddlerMaRdw2J1h6Lfc

Figure 1: Comparison of `crypt()` and `WSJ.com` authentication cookies. The last field represents the username prepended to the output of the `crypt()` function. The input to the `crypt()` function is the username prepended to the string "March20".

Prohibit guessable passwords

Many Web sites advise users to choose memorable passwords such as birthdays, names of friends or family, or social security numbers. This is extremely poor advice, as such passwords are easily guessed by an attacker who knows the user. Even without bad advice, passwords are fairly guessable [33]. Thus, servers ought to prohibit users from using any password found in a dictionary; such passwords are vulnerable to dictionary attacks. Servers can reduce the effectiveness of on-line dictionary attacks by restricting the number of failed login attempts or requiring a short time delay between login attempts.

Unfortunately, implementing this requirement will make a Web site less appealing to use since it makes passwords harder to remember.

Reauthenticate before changing passwords

In security-sensitive operations such as password changing, a server should require a user to reauthenticate. Otherwise, it may be possible for an adversary to replay an authentication token and force a password change, without actual knowledge of the current password.

3.3 Handle authenticators carefully

Authenticators are the workhorse of any authentication scheme. These are the tokens presented by the client to gain access to the system. As discussed above, authenticators protect passwords by being a short-term secret; the authenticator can be changed at any time whereas passwords are much less convenient to change.

Make authenticators unforgeable

Many sites have authenticators that are easily predictable. For instance, we noticed that `highschoolalumni.com` uses ID numbers and email addresses inside cookies to authenticate users. An interrogative adversary can find this information in the publicly available alumni database, and mint an authenticator for an any user.

Authenticators often contain keys that function as session identifiers. These identifiers should be cryptographically random; statistical randomness is not sufficient. The Allaire Cold Fusion Web server issues CFTOKEN session identifiers which come from a linear congruential number generator [2]. As described above, `FatBrain.com` used essentially a global sequence number. While these numbers may be appropriate for tracking users, it is possible for an adversary to deduce the next output, and hence the next valid session identifier. This may allow the adversary access the information of another user.

Authenticators may also contain other information that the system will accept to be true. Thus, they must also be protected from tampering. This is done by use of a message authentication code (MAC). Because message authentication codes require a secret key, only an entity with knowledge of the key can recreate a valid code. This makes the codes unforgeable since no adversary should possess the secret key. Use only strong cryptographic hash functions. Do not use CRC codes or other non-cryptographic hashes, as such functions are often trivial to break.

Relatedly, when combining multiple pieces of data to input into a message authentication code, be sure to unambiguously separate the components. Since most inputs are text, this can be done using some character that is known not to appear in the input fragments. If components are not clearly separated, multiple inputs can lead to the same outputs. For example, "usernameaccess" could come from "username" followed by "access" or "user" followed by "nameaccess"; better to write "username&access" to ensure that the interpretation is unambiguous. Of course, care must be taken to prevent the username from containing an ampersand!

Protect authenticators that must be secret

Some systems believe that they are secure against eavesdropping adversaries because they send their authenticators over SSL. However, a secure transport is ineffective if the authenticators leak through plaintext channels. We describe two ways that authenticators are sent over SSL and mistakes which can lead to the authenticator leaking into plaintext.

One method is to set the authenticator as a cookie. When doing so, it is usually appropriate to set the Secure flag on cookies sent over SSL. When set to true, this flag instructs a Web browser to send the cookie over SSL only. A number of SSL Web sites neglect to set this flag. This simple error can completely nullify the useful properties of SSL. For instance, customers of SprintPCS can view their account information and make equipment purchases online. To authenticate, a user enters a phone number and password over SSL. SprintPCS then sets a cookie which acts as an authenticator. Anyone with the cookie can log in as that user. The protocol so far is reasonably secure. However, because SprintPCS does not set the Secure flag on their authentication cookie, the authenticator travels in plaintext over HTTP whenever a user visits the main SprintPCS Web page. We believe that SprintPCS intended to protect against eavesdropping adversaries. Nevertheless, a eavesdropping adversary can access a victim's account with a replay because the cookie authenticator leaks over HTTP.

A second method of setting an authenticator is to include it as part of the URL. Though the HTTP 1.1 specification [14] recommends against this, it easy to do and sites still use this. The problem with this method is that it too can leak authenticators through plaintext channels. If a user follows a link from one page to another, the Web browser usually sends the Referer [sic] header. This field includes the URL of the page from which the current request originated. As described in Section 14.36 of the HTTP specification, the Referer field is normally used to allow a server to trace back-links for logging, caching, or maintenance purposes. However, if the URL of the linking page includes the authenticator, the server will receive a copy of the authenticator in the HTTP header. Section 15.1.3 of the specification recommends that clients should not include a Referer header in a non-secure HTTP request if the referring page was transferred with a secure protocol for exactly this reason. However, this is not a requirement; browsers such as Netscape and Lynx send the Referer header anyway without any warning.

This can be exploited via a cross-site scripting attack [9]. An adversary can cause a user to execute arbitrary code and offer the user a link from a secure URL including the authenticator (that appears legitimate) to a link of the adversary's choosing. If the user selects the link, the Referer field in the request may include the authenticator, making it available to a eavesdropping adversary. Worse, the link could point to the adversary's machine. Then no eavesdropping is necessary to capture the authenticator. If the attacker is clever and uses an SSL server to host the attack, most browsers will not indicate that anything untoward is happening since they only warn users about transitions from SSL to non-SSL links.

Therefore, be careful when setting authenticators in cookies and follow the recommendation of the HTTP 1.1 specification by not using authenticators in URLs.

Avoid using persistent cookies

A *persistent* cookie is written to a file on the user's system; an *ephemeral* or *temporary* cookie is only stored in the browser's memory and disappears when the user exits the browser. An error in the way the browser or user handles the cookie file may make it accessible over the Internet, exposing the user's cookies to anyone who knows where to look. For instance, certain queries to search engines can produce many cookie files accidentally placed on the Web [18]. If a persistent cookie in a leaked file contains an authenticator, an adversary can simply copy the cookie and break into the user's account. In addition, if the user accesses the account from a public system (say at a library or Internet café) and receives a persistent authentication cookie on that system, any subsequent user of that system can access the account. For these reasons, persistent cookies should not be considered private. Do not store authenticators in persistent cookies.

Limit the lifetime of authenticators

A good design must also gracefully handle the compromise of tokens which are designed to be secret. To limit the amount of damage a leaked authenticator can cause, limit its lifetime.

For authenticators that are stored in user cookies, do not rely on the cookie expiration field for secure expiration. Since the client is responsible for enforcing that expiration, a malicious client can set the lifetime arbitrarily. Netscape users can manually extend these expirations by simply editing a text file. We were able to indefinitely extend the lifetime of our `WSJ.com` cookie authenticator even though `WSJ.com` set the cookie to expire in 11 hours. This was not extremely alarming, but if an adversary stole a cookie (as described in Section 3.3), there would be no way to revoke the adversary's access. The problem was compounded because the cookie authenticator remained the same even if a user's password changed. This prevented the `WSJ.com` site from easily revoking access to a compromised account.

To prevent unauthorized cookie lifetime extensions, include a cryptographically unalterable timestamp in the value of the cookie, or store the expiration time in a user-inaccessible place on the server. Securely binding expirations to authenticators limits the damage caused by a stolen authenticator.

Note that an authenticator that is stored in a cookie can be replayed, regardless of its expiration time, if it is leaked. By definition, unless the client uses computation, the only thing it is capable of doing to the cookie is to send it back to the server. If replay prevention is desired, the authenticator must be kept confidential and changed after each use. In that case, it might be necessary to record recently received authenticators and verify that newly received authenticators are not replays.

Bind authenticators to addresses

It can also be useful to tie authenticators to specific network addresses. This helps protect against replay attacks by making it more difficult for the adversary to successfully reuse the authenticator. In addition to acquiring the authenticator, the adversary must appear to originate from the same network address for which the authenticator was minted. However, this may prematurely invalidate authenticators issued to mobile DHCP users.

4 Design

In this section we present a scheme for performing client authentication. This design is intended to be an example of a simple system that follows the hints provided in Section 3. We do not claim that the scheme is novel, but we do claim that the concepts and design process are not extensively discussed in literature. We present a brief security analysis of the schemes in Section 5.

Our scheme provides a personalizable authenticator which allows the server to statelessly verify the authenticity of the request and its contents. The server can explicitly control the valid lifetime of the authenticator as well. The authenticator can include all the information needed to service a request, or can be used as a key to refer to session information stored on the server.

The overall operation of this scheme is shown in Figure 2. We assume that the user has an existing account on the server which is accessed via a username and password. At the start of each session, the server receives the username and password, verifies them, and sets an authentication cookie on the user's machine. Since cookies are widely supported, this makes the system portable. Subsequent requests to the server include this cookie and allow the server to authenticate the request. The design of each cookie ensures that a valid cookie can only be created by the server; therefore anyone possessing a valid cookie is authorized to access the requested content on the server.

Our scheme is designed to be secure against an interrogative adversary, as we believe that most of the schemes we evaluated were designed with this type of adversary in mind. However, because SSL with server authentication provides confidentiality and integrity, layering our design on top of SSL can provide an authentication system secure against an active adversary.

4.1 Cookie Recipe

The recipe for our cookie follows easily from the hints presented in Section 3. We create an unforgeable authenticator that includes an explicit expiration time. We use HTTP state (i.e. cookies) to store this authenticator with the client. The value of this cookie is shown here:

$$\texttt{exp=}t\texttt{\&data=}s\texttt{\&digest=MAC}_k\texttt{(exp=}t\texttt{\&data=}s\texttt{)}$$

The expiration time is denoted t and is expressed as seconds past 1970 GMT. The data string s is an optional parameter denoting arbitrary data that the server wishes to associate with the client. Finally, the cookie includes a MAC for the cleartext expiration and data.

Our cookie requires the use of a *non-malleable* MAC; that is, one where it is intractable to generate a valid ciphertext from a plaintext message related to a plaintext message with a known ciphertext [12, 24]. That is, no adversary can generate a valid ciphertext without both the server's secret key and the plaintext, no matter how many samples of valid plaintext/ciphertext pairs the adversary has. Examples of keyed, non-malleable MACs are HMAC-MD5 and HMAC-SHA1 [24].

4.2 Discussion

Selecting an expiration time t is a trade-off between limiting the damage that can be done with a leaked authenticator and requiring the user to reauthenticate. Yahoo!, for example, allows users to specify what expiration interval they prefer for authenticators that control access to sensitive data [47]. This allows the user to control the trade-off. On the other hand, for insensitive data, it makes sense for the server to make the choice. For example, a newspaper might want cookies to be valid for

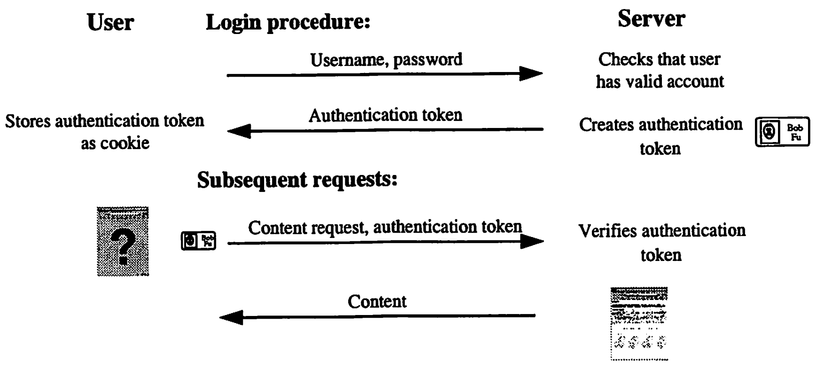

User
Login procedure:
Server
Username, password
Checks that user has valid account
Stores authentication token as cookie
Authentication token
Creates authentication token
Subsequent requests:
Content request, authentication token
Verifies authentication token
Content

Figure 2: One-exchange authentication system.

only a day, whereas a magazine might allow sessions to be valid for a month (as if the user were buying a single issue).

The value s may be any information specific to the user that the server wishes to access without maintaining server state. This may be anything from a session identifier to a username. Beware that this data is not encrypted so sensitive information should not be stored here; if sensitive data is needed, we recommend that a cryptographically random session identifier be used. This will prevent information leaks from compromising a user's privacy. On the other hand, if sensitive user information is required to handle only a small percentage of the content requests, the authenticator can contain the information needed to service the majority of requests. This way the server can avoid doing a possibly expensive look-up with every request.

A server may also choose to leave s empty (and removing the `data` parameter from the cookie). This might be useful in the case where authentication must expire, but all users are essentially the same. A plausible example of this might be a pay-per use service, such as a newspaper.

4.3 Authentication and revocation

To authenticate a user, the server retrieves the cookie and extracts the expiration. If the cookie has not expired, the server recalculates the MAC in the `digest` parameter of the cookie. Since the server is the only entity who knows the key k, the properties of the MAC function imply that a valid cookie was generated by the server. So long as the server only generates cookies for authenticated users, any client with a valid cookie is a valid user.

This scheme does not provide a mechanism for secure *revocation*; that is, ending the user's session before the expiration time is up. The easiest option is for the server can instruct the client to discard the authentication cookie. This will usually be adequate for most applications. However, a client who has saved the value of the cookie can continue to reuse that value so long as the explicit expiration time has not yet passed.

In most cases, a short session can make revocation unnecessary: the user can access the server until the session expires, at which time the server can refuse to issue a new authenticator. Servers that require secure revocation should keep track of the session status on the server (e.g., using a random session key or our personalized scheme with a server database). This session can then be explicitly revoked on the server, without trusting the client.

The scheme does allow simultaneous revocation of all authenticators, which can be accomplished by rotating the server key. This will cause all outstanding cookies to fail to verify. Thus, all users will have to log in again. This might be useful for finding unused accounts.

4.4 Design alternatives

One interesting point of our scheme is that we have included the expiration time t in the cookie value itself. This is the only way for a server to have access to the expiration date without maintaining state. Explicit inclusion of the expiration date in a non-malleable cookie provides fixed-length sessions without having to trust the client to expire the cookie. It would also have been possible to merely use a session identifier but that would always require server state and might lead to mistakes where expiration was left in the hands of the client.

Many schemes do involve setting a random session identifier for each user. This session identifier is used to access the user's session information, which is stored in a database on the server. While such a scheme allows for a client to make customizations (i.e. it is functionally equivalent to the scheme we have presented), it is potentially subject to guessing attacks on the session identifier space. If an adversary can successfully guess a session identifier, the system is broken (see Section 3.3). Our scheme provides a means for authenticating clients that is resistant to guessing attacks on session identifiers. Furthermore, our scheme provides the option of authenticating clients with $O(1)$ server state, rather than $O(n)$, where n is the number of clients.

Our system can also make it easier to deploy multi-server systems. Using session identifiers requires either synchronized, duplicated data between servers or a single server to coordinate requests, which becomes a potential bottleneck. Our scheme allows any server to authenticate any user with a minimum of information, none of which must be dynamically shared between servers. In addition, the authentication always completes in constant time, rather than in time which increases with the number of users.

5 Security analysis

In this section we present an informal analysis of the security properties of our design. For the purpose of discussion, we will refer to the cookie's two halves: the *plaintext* and the *verifier*. The plaintext is the expiration concatenated with the user string, and the verifier is the HMAC of the plaintext.

We will discuss the security of the scheme once the authenticator (i.e. cookie) is received by the user from the server. We will not discuss mechanisms for completing the initial login.

5.1 Forging authenticators

An adversary does not need to log in if it can create a valid authenticator offline. Often an adversary can create a plausible plaintext string; therefore the security of the authenticator rests on the fact that the verifier cannot be calculated by an adversary without the key. Since we have selected our MAC to be non-malleable, an adversary can not forge a new authenticator.

An attacker may also attempt to extend the capabilities associated with the authenticator. This might include changing the expiration date or some aspect of the data string which would allow unauthorized access to the server. For instance, if the data string includes a username, and the adversary can alter the username, this might allow access another user's account. It is easy enough for the adversary to change the plaintext of the authenticator in the desired manner. However, as we have seen, because HMAC is non-malleable, it is intractable for the adversary to generate a valid ciphertext for an altered plaintext string. Therefore the adversary cannot bring about any change in an authenticator that will be accepted by the server.

5.2 Authenticator hijacking

An interrogative adversary cannot see any messages that pass between the user and the server. Therefore, it cannot hijack another user's authenticator. However, an eavesdropper can see the authenticator as it passes between the user and the server. Such an adversary can easily perform a replay attack. Therefore the system is vulnerable to hijacking by such an adversary. However, the replay attack lasts only as long as the authenticator is valid; that is, between the time the adversary "sniffs" the authenticator and the expiration time. The adversary does not have the ability to create or modify a valid authenticator. Therefore this is an attack of limited usefulness. The lifetime of the authenticator determines how vulnerable the system is; systems which employ a shorter authenticator lifetime will have to reauthenticate more often, but will have tighter bounds on the damage that a successful eavesdropping adversary can accomplish. In addition, the system can protect against an eavesdropping adversary by using SSL to provide confidentiality for the authenticator.

5.3 Other attacks

We mention briefly some attacks on our schemes which do not deal with the authenticator directly. The best known attack against the scheme in Section 4 is a brute force key search.

A server compromise breaks the system: if the adversary obtains the key to the MAC, it can generate valid authenticators for all users. Random keys and key rotation help to prevent the adversary from mounting brute force key attacks (see Lenstra [28] for suggestions on key size).

In addition, key rotation helps protect against volume attacks, whereby an adversary may be able to obtain the key to the hash function because the adversary has obtained a great quantity of data encrypted using it. We note that HMAC-MD5 and HMAC-SHA1 are not believed to be vulnerable to this type of analysis [24]. However, we believe that it is prudent to include key rotation since it does not decrease the security of the scheme, it protects against server compromise, and it has minimal cost to the server.

In addition, the adversary can obtain unauthorized access by guessing the user's password; see Section 3.2 for some guidelines for preventing this.

Our scheme in itself only provides user authentication. For protection against server impersonation or for data integrity, we recommend SSL.

6 Implementation and performance

The client authentication scheme described in Section 4 was implemented in Perl 5.6 using the LWP, HTTP, CGI, FCGI, and Digest modules. We tested the implementation on two dual Pentium III 733 MHz machines each with 256 MB of RAM running the Linux 2.2.18-smp kernel and Apache 1.3.17 with mod_fastcgi 2.2.10. Everything ran on a local disk. A dedicated Gigabit link with a 20 μs round-trip time connected the machines.

6.1 Microbenchmark performance

We ran 1,000 trials of `crypt()` and HMAC-SHA1. The input to `crypt()` was an 8-byte input and a 2-byte salt. The input to HMAC-SHA1 was a 27-byte input and a 20-byte key. `crypt()` finished on average in 8.08 μsec with 99% of the trials completing in under 10 μsec. HMAC-SHA1 took on average 41.4 μsec with 99% of the trials completing in under 47 μsec. We attribute the variances to context switching.

6.2 End-to-end performance

To measure the end-to-end performance of cookie-based logins, we repeatedly retrieved 400 bytes of data from a Web server that authenticated our client. Both the client and the cookie authentication scheme were implemented in Perl, and the server ran the cookie authentication script with FastCGI. Our end-to-end test consisted of the client presenting a cookie authenticator (as described in Section 4) to the server, which verifies the authenticator by performing HMAC-SHA1 on the expiration date presented by the client. In order to provide a baseline for comparison, we also measured the average performance of plain HTTP, HTTP with Basic Authentication [16], and an always-authenticated FastCGI script for the same page.

For each scheme, we made 5,000 successive requests, with valid authentication information (when needed).

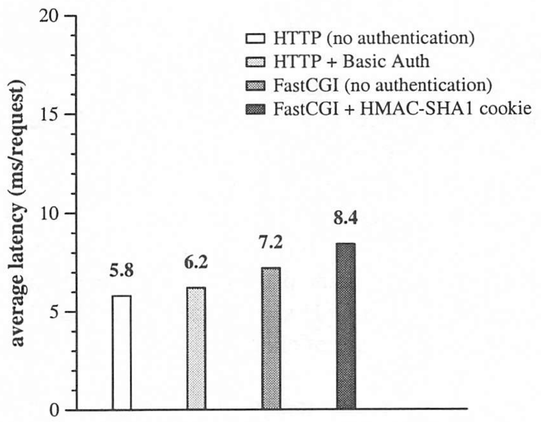


Figure 3: End-to-end performance of average service latency per request. We measure HTTP and FastCGI without authentication to obtain a baseline for comparison. Basic Auth is the cleartext password authentication in HTTP [16].

Figure 3 presents the average time from the request being sent in our HTTP client until a response was received.

99% of the HTTP trials without authentication were faster than 5.9 ms. Similarly, 99% of HTTP Basic authentication trials were faster than 6.3 ms. 99% of the plain FastCGI trials were faster than 7.7 ms, and 99% of the FastCGI trials with our HMAC-SHA1 scheme took less than 8.8 ms. Figure 3 shows that the cost of HTTP Basic authentication is 0.4 ms per request while the cost of our HMAC-SHA1 scheme is 1.2 ms. We suspect that non-cryptographic factors such as string parsing and file I/O cause the disparity between the microbenchmarks and the end-to-end measurements.

Note that SSL is an order of magnitude slower than the HMAC-SHA1 cookie scheme. A single new SSL connection takes 90 ms [17] on a reasonable machine. SSL client authentication, even with session resumption, cannot run faster than the HMAC-SHA1 cookie scheme because SSL authenticates the entire HTTP stream. Our scheme runs HMAC-SHA1 on fewer than 30 bytes of data per request (a timestamp, personalization data, and a key).

7 Related work

There is an extensive body of work related to authentication in general and Web authentication in particular. We highlight a few relevant examples. For other studies of design principles, see Abadi [1] or Lampson [27].

7.1 General authentication protocols

In the past ten years, several new authentication protocols have been developed, including AuthA [4], EKE [5], provably secure password authenticated key exchange [7], and the Secure Remote Password protocol [45]. Furthermore, groups are simplifying and standardizing password authentication protocols [22]. However, these protocols are not well-suited for the Web because they are designed for session initialization of long-running connections, as opposed to the many short-lived connections made by Web browsers. Long-running connections can easily afford a protocol involving the exchange of multiple messages, whereas short-lived ones cannot absorb the overhead of several extra round-trips per connection. Additionally, these protocols often require significant computation, making them undesirable for loaded Web servers.

One-time passwords can prevent replay attacks. Lamport's user password authentication scheme defends against an adversary who can eavesdrop on the network and obtain a copies of server state (i.e. the hashed password file) [26]. This scheme is based on a one-way function. Haller later implemented the S/Key one-time password system [20, 21] using techniques from Lamport. De Waleffe and Quisquater extended Lamport's scheme with zero-knowledge techniques to provide more general access control mechanisms [10]. With their one-exchange protocol, a user can authenticate and prove possession of a ticket. This scheme is not appropriate for our model of Web client authentication because it requires the client to perform computation such as modular exponentiation.

Kerberos uses tickets to authenticate users to services [23, 34, 41]. The Kerberos ticket is encrypted with a key known only to the service and the Kerberos infrastructure itself. A temporary session key is protected by encryption. The ticket approach differs greatly from schemes such as ours because tickets are message preserving, meaning that an adversary who compromises a service key can recover the session key. If an adversary compromises the key in our scheme, it can mint and verify tokens, but it cannot recover the contents that were originally authenticated. Authentication and encryption should be separated, but Kerberos does both in one step.

The Amoeba distributed operating system cryptographically authenticated capabilities (or rights) given to a user [43]. One of the proposed schemes authenticated capabilities by XORing them with a secret server key and hashing the result. Client authentication on the Web falls into the same design space. A Web server wishes to send a user a signed capability.

7.2 Web-specific authentication protocols

The HTTP specifications provide two mechanisms for authentication: Basic authentication and Digest authentication [16]. Basic authentication requires the client to send a username and password in the clear as part of the HTTP request. This pair is typically resent preemptively in all HTTP requests for content in subdirectories of the original request. Basic authentication is vulnerable to an eavesdropping adversary. It also does not provide guaranteed expiration (or logout), and repeatedly exposes a user's long-term authenticator. Digest authentication, a newer form of HTTP authentication, is based on the same concept but does not transmit cleartext passwords. In Digest authentication, the client sends a cryptographic hash (usually MD5) of the username, password, a server-provided nonce, the HTTP method, and the URL. The security of this protocol is extensively discussed in RFC 2617 [16]. Digest authentication enjoys very little client support, even though it is supported by the popular Apache Web server.

The main risk of these schemes is that a successful attack reveals the user's password, thus giving the adversary unlimited access. Further, breaks are facilitated by the existence of freely available tools capable of sniffing for authentication exchanges [40].

The Secure Sockets Layer (SSL) protocol is a stronger authentication system provides confidentiality, integrity, and optionally authentication at the transport level. It is standardized as the Transport Layer Security protocol [11]. HTTP runs on top of SSL, which provides all the cryptographic strength. Integration at the server allows the server to retrieve the authentication parameters negotiated by SSL. SSL achieves authentication via public-key cryptography in X.509 certificates [8] and requires a public-key infrastructure (PKI). This requirement is the main difficulty in using SSL for authentication — currently there is no global PKI, nor is there likely to be one anytime soon. Several major certificate authorities exist (e.g., Verisign), but the space is fractured and disjoint. To some degree, users avoid client certificates because certificates are practically incomprehensible to non-technical users. Other arguments suggest that the merits of PKI as the answer to many network security problems have been somewhat exaggerated [13]. Client support for SSL is non-standard and thus can have interoperability problems (e.g., Microsoft Internet Explorer and Netscape Navigator client certificates do not interoperate), and performance concerns. SSL decreases Web server performance and often provides more functionality than most applications need. In an effort to avoid using SSL, Bergadano, Crispo, and Eccettuato use Java applets to secure HTTP transactions [6].

Park and Sandu identify security problems of regular cookies, network threats, end-system threats, and cookie harvesting threats [35]. Samar describes a cookie-based distributed architecture for single-signon [38].

7.3 Schemes in the field

Many ad hoc schemes are used today to perform Web authentication without making use of either SSL or any of the HTTP authentication mechanisms. Instead, schemes often use HTTP state management to store authenticators with the client. This helps sites provide authentication for Web applications while preserving ease-of-use and performance. While many of these schemes are well-designed and do indeed provide appropriately strong authentication for the environment in which they are deployed, just as many schemes have fatal flaws.

Shibboleth, a project of Internet2, is investigating architectures, frameworks, and technologies to support cross-realm authentication and authorization for access to Web pages [39]. The group completed a survey of client authentication on the Web at several universities, most of which use a combination of Kerberos, client certificates, HTTP authentication, and cookies. However, they have not yet presented a complete design.

Open Market has patented a scheme that creates a folded cryptographic hash of a server secret, a session identifier, and other parameters [29]. Yahoo has a cookie authentication scheme that computes MD5 of a server secret, user identifier, timestamp, and other parameters [47]. This scheme is documented on our Web site. The ArsDigita Community System (ACS) has a SHA1-based cookie authentication scheme [30]. All these schemes are likely to be secure against interrogative adversaries, but all appear vulnerable to eavesdroppers.

Microsoft Passport offers a managed cookie authentication scheme [36]. Microsoft mints a cookie authenticator after a user logs in. Vendors participating in the passport service can verify the authenticator to determine authenticity and authorization. The details of the authentication scheme have not been published, but the white paper indicates that Microsoft shares a unique symmetric key with each vendor. These keys can both mint and verify authenticators.

8 Conclusion

To provide designers and implementers with a clear framework, we have given a description of the limitations, requirements, and security models specific to Web client authentication. We presented a set of hints on how to design a secure client authentication scheme, based on experience gained from our informal survey of commercial schemes. The survey showed that many sites are not secure against the interrogative adversary. We proposed an authentication scheme secure against the interrogative adversary.

Web sites have such a large range of requirements that no one authentication scheme can meet them all. Currently SSL remains too costly and client authentication infrastructures remain hardly deployed. This partially explains why so many home-brew schemes exist. The Web community ought to recommend a secure standard or secure practices if there is any hope to eliminate the proliferation of insecure home-brew authentication schemes. We hope that this paper will help schemes in resisting common attacks.

For more information and our source code, download our technical report [18] or visit our Web site at `http://cookies.lcs.mit.edu/`.

9 Acknowledgments

We thank David Andersen, Ian Anderson, Jeffrey W. Baker, Richard Barbalace, Andrew M. Boardman, Benjie Chen, David Dittrich, Paul Hill, Frans Kaashoek, David Mazières, Robert T. Morris, Steve Morris, Joon Park, Matt Power, Ron Rivest, Jerry Saltzer, Richard Smith, Win Treese, the anonymous reviewers, and the members of the PDOS group at MIT. We also thank the companies who talked with us about the security of their Web sites: `FatBrain.com`, `WSJ.com`, and `yahoo.com`. The students of the MIT Applied Security Reading Group (`http://pdos.lcs.mit.edu/asrg/`) deserve credit for the genesis of this project. Finally, we thank Duncan Hines for manufacturing the materials necessary to sustain our efforts.

References

[1] Martín Abadi and Roger Needham. Prudent engineering practice for cryptographic protocols. Technical Report 125, DEC Systems Research Center, June 1994.

[2] Allaire Corporation. Personal Communication, January 2001.

[3] Mihir Bellare, Anand Desai, David Pointcheval, and Phillip Rogaway. Relations among notions of security for public-key encryption schemes. In Hugo Krawczyk, editor, *Proceedings of Advances in Cryptology—CRYPTO 98*, volume 1462 of *Lecture Notes in Computer Science*, pages 26–45, Santa Barbara, CA, 1998. Springer-Verlag.

[4] Mihir Bellare and Phillip Rogaway. The AuthA protocol for password-based authenticated key exchange. Technical report, IEEE P1363, March 2000. `http://grouper.ieee.org/groups/1363/StudyGroup/Passwd.html#autha`.

[5] Steven M. Bellovin and Michael Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *Proceedings of the 1992 IEEE Symposium on Security and Privacy*, pages 72–84, Oakland, CA, May 1992.

[6] F. Bergadano, B. Crispo, and M. Eccettuato. Secure WWW transactions using standard HTTP and Java applets. In *Proceedings of the 3rd USENIX Workshop on Electronic Commerce*, pages 109–119, Boston, MA, September 1998.

[7] Victor Boyko, Philip MacKenzie, and Sarvar Patel. Provably secure password authenticated key exchange using Diffie-Hellman. In B. Preneel, editor, *Proceedings of Advances in Cryptology—EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, Bruges, Belgium, May 2000. Springer-Verlag.

[8] CCITT. Recommendation X.509: The directory authentication framework, 1998.

[9] CERT. Malicious HTML tags embedded in client Web requests. CA-2000-02, February 2000. `http://www.cert.org/advisories/CA-2000-02.html`.

[10] Dominique de Waleffe and Jean-Jaques Quisquater. Better login protocols for computer networks. In B. Preneel, R. Govaerts, and J. Vandewalle, editors, *Proceedings of Computer Security and Industrial Cryptography*, volume 741 of *Lecture Notes in Computer Science*, pages 50–70. Springer-Verlag, 1993.

[11] Tim Dierks and Christopher Allen. The TLS protocol version 1.0. RFC 2246, Network Working Group, January 1999.

[12] Danny Dolev, Cynthia Dwork, and Moni Naor. Nonmalleable cryptography. In *Proceedings of the 23rd ACM Symposium on Theory of Computing*, pages 542–552, New Orleans, LA, 1991.

[13] Carl Ellison and Bruce Schneier. Ten risks of PKI: What you're not being told about public key infrastructure. *Computer Security Journal*, 16(1):1–7, 2000.

[14] Roy Fielding, James Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext Transfer Protocol — HTTP/1.1. RFC 2616, Network Working Group, June 1999.

[15] FIPS 180-1. *Secure Hash Standard*. U.S. Department of Commerce/N.I.S.T., National Technical Information Service, Springfield, VA, April 1995.

[16] John Franks, Phillip Hallam-Baker, Jeffrey Hostetler, Scott Lawrence, Paul Leach, Ari Luotonen, and Lawrence Stewart. HTTP authentication: Basic and digest access authentication. RFC 2617, Network Working Group, June 1999.

[17] Kevin Fu, M. Frans Kaashoek, and David Mazières. Fast and secure distributed read-only file system. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000)*, pages 181–196, San Diego, CA, October 2000.

[18] Kevin Fu, Emil Sit, Kendra Smith, and Nick Feamster. Dos and don'ts of client authentication on the Web. Technical Report 818, MIT Laboratory for Computer Science, May 2001. `http://www.lcs.mit.edu/`.

[19] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal of Computing*, 17(2):281–308, April 1988.

[20] Neil Haller. The S/KEY one-time password system. In *Proceedings of the ISOC Symposium on Network and Distributed System Security*, pages 151–157, San Diego, CA, February 1994.

[21] Neil Haller. The S/KEY one-time password system. RFC 1760, Network Working Group, February 1995.

[22] IEEE P1363a: Standard specifications for public key cryptography: Additional techniques. `http://www.manta.ieee.org/groups/1363/P1363a`.

[23] John T. Kohl. The use of encryption in Kerberos for network authentication. In G. Brassard, editor, *Proceedings of Advances in Cryptology—CRYPTO 89*, volume 435 of *Lecture Notes in Computer Science*, pages 35–43. Springer-Verlag, 1990.

[24] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. HMAC: Keyed-hashing for message authentication. RFC 2104, Network Working Group, February 1997.

[25] David Kristol and Lou Montulli. HTTP State Management Mechanism. RFC 2965, Network Working Group, October 2000.

[26] Leslie Lamport. Password authentication with insecure communication. *Communications of the ACM*, 24(11):770–771, November 1981.

[27] Butler Lampson. Hints for computer system design. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pages 33–48, Bretton Woods, NH, 1983.

[28] Arjen Lenstra and Eric Verheul. Selecting cryptographic key sizes. `http://www.cryptosavvy.com/cryptosizes.pdf`, November 1999.

[29] Thomas Levergood, Lawrence Stewart, Stephen Morris, Andrew Payne, and Winfield Treese. Internet server access control and monitoring systems. U.S. patent #5,708,780, Open Market, January 1998.

[30] Richard Li and Archit Shah. ArsDigita Community System (ACS) security design. `http://developer.arsdigita.com/doc/security-design.html`.

[31] Alfred J. Menezes, Paul C. Van Oorschot, and Scott A. Vanstone. *Handbook of applied cryptography*. The CRC Press series on discrete mathematics and its applications. CRC Press, 1997.

[32] Keith Moore and Ned Freed. Use of HTTP State Management. RFC 2964, Network Working Group, October 2000.

[33] Robert Morris and Ken Thompson. Password security: A case history. *Communications of the ACM*, 22(11):584–597, November 1979.

[34] B. Clifford Neuman and Theodore Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications Magazine*, 32(9):33–38, September 1994.

[35] Joon S. Park and Ravi Sandhu. Secure cookies on the Web. *IEEE Internet Computing*, 4(4):36–44, July/August 2000.

[36] Microsoft passport. `http://www.passport.com/`.

[37] Eric Rescorla. *SSL and TLS: Designing and Building Secure Systems*. Addison-Wesley, 2000.

[38] Vipin Samar. Single sign-on using cookies for Web applications. In *Proceedings of the 8th IEEE Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 158–163, Palo Alto, CA, 1999.

[39] The Shibboleth Project. `http://middleware.internet2.edu/shibboleth/`.

[40] Dug Song. *dsniff*. `http://www.monkey.org/~dugsong/dsniff/`.

[41] Jennifer Steiner, Clifford Neuman, and Jeffrey Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the Winter 1988 USENIX*, pages 191–202, Dallas, TX, February 1988.

[42] Paul Syverson, Stuart Stubblebine, and David Goldschlag. Unlinkable serial transactions. In R. Hirschfeld, editor, *Proceedings of Financial Cryptography*, volume 1318 of *Lecture Notes in Computer Science*, Anguilla, BWI, 1997. Springer-Verlag.

[43] Andrew Tanenbaum, Sape Mullender, and Robbert van Renesse. Using sparse capabilities in a distributed system. In *Proceedings of the 6th International Conference on Distributed Computing*, pages 558–563, Cambridge, MA, 1986.

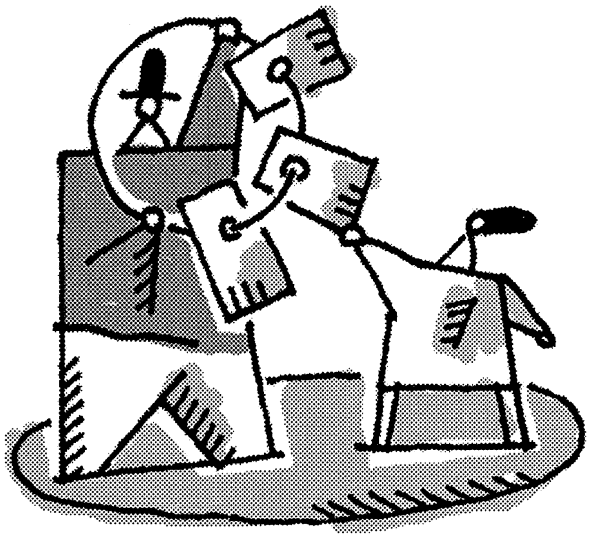
[44] David Wagner and Ian Goldberg. Proofs of security for the Unix password hashing algorithm. In T. Okamoto, editor, *Proceedings of Advances in Cryptology—ASIACRYPT 2000*, volume 1976 of *Lecture Notes in Computer Science*, Kyoto, Japan, December 2000. Springer-Verlag.

[45] Thomas Wu. The secure remote password protocol. In *Proceedings of the 1998 Internet Society Network and Distributed System Security Symposium*, pages 97–111, San Diego, CA, March 1998.

[46] Web and mobile code security. `http://www.securityfocus.com/forums/www-mobile-code/`.

[47] Yahoo, Inc. Personal Communication, November 2000.

KEY MANAGEMENT



Session Chair: Peter Gutmann, *University of Aukland*

SC-CFS: Smartcard Secured Cryptographic File System

Naomaru Itoi

Center for Information Technology Integration
University of Michigan
Ann Arbor

`http://www.citi.umich.edu/projects/smartcard/`

Storing information securely is one of the most important roles expected for computer systems, but it is difficult to achieve with current commodity computers. The computers may yield secrets through physical breach, software bug exploitation, or password guessing attack. Even file systems that provide strong security, such as the cryptographic file system, are not perfect against these attacks. We have developed SC-CFS, a file system that encrypts files and takes advantage of a smartcard for per-file key generation. SC-CFS counters password guessing attack, and minimizes the damage caused by physical attack and bug exploitation. The performance of the system is not yet satisfactory, taking 300 ms for accessing a file.

1 Introduction

Storing information securely has been one of the most important applications of computer systems since their introduction. As information technology is being integrated into society rapidly, secure storage is now demanded more strongly, and by more people, than ever. For example, consider the recent incident in which a laptop computer was stolen from the State Department of the United States in January 2000 [20]. Not only the people who deal with highly classified information, but also ordinary people are threatened by hackers, as they store their private data on computers today, e.g., e-mail, financial information, Internet activity history, and medical history.

For the purpose of this paper, we define *secure storage* as "a storage system that protects the secrecy, authenticity, and integrity of the information it stores".[1] Unfortunately, modern commodity computers cannot provide secure storage because of the three prevalent, but inaccurate, assumptions about computer systems. First, modern commodity computers tend to overlook physical security, and lack physical protection; read and write access to computational and storage devices is typically possible by simply opening the cover of a computer. For example, a hard disk drive is easily removed, giving full access to an adversary. Second, bugs in design and implementation of software are unavoidable [8], and can be exploited to give away secrets. Exploitable bugs are found in all ranges of software, and some of them are so serious that they lead to administrative rights (`root`) compromise [11]. Third, passwords are often the weakest link in security systems. Once passwords are stolen, no matter how securely the system is designed and implemented, it becomes vulnerable to impersonation. Passwords can be stolen from memory, from virtual memory backing store [23], in transit through networks [25], or can be guessed with dictionary attack [17].

An obvious countermeasure to theft of secrets is to encrypt the secrets with an encryption key, and protect the key. Matt Blaze has realized this with a *Cryptographic File System for UNIX (CFS)* [2], which transparently encrypts files in a file system [4]. Although CFS adds significant security to current systems, it still suffers from the the problems introduced above. First, CFS relies on user chosen passwords to provide encryption keys, making dic-

[1] Denning defines the desired properties of a communication channel similarly [6].

[2] Throughout this document, we refer to CFS version 1.3.3 by "CFS".

tionary attack possible. An adversary can obtain ciphertext through physical attack or bug exploitation, and can run an off-line dictionary attack. Second, the number of passwords a user can remember is limited. To lower the burden of the user, CFS uses one key to encrypt all the files in a directory tree, which is not as desirable as using one key for each file. If the key is stolen, physically or through exploitation, the files encrypted under the key are revealed. Therefore, the fewer files are encrypted under a single key, the better.

We attack this problem by storing a randomly generated user key on a smartcard, and generating a file key that is used to encrypt only one file. We have implemented such a system called SC-CFS, based on CFS. Instead of a password, SC-CFS uses the random key on a smartcard to generate file keys, thus thwarting dictionary attack. On host compromise, SC-CFS reveals only the keys of the files that are currently used (and these files are already in memory in the clear, anyway), thus minimizing damage. The design, security considerations, implementation, performance evaluation, related work, and future direction are discussed in this paper.

2 Design

2.1 Cryptographic File System Review

As SC-CFS is based on CFS, it is important to understand how CFS works. CFS consists of a CFS daemon, or *cfsd*, and application programs. `cfsd` is a *Network File System* [21] server daemon (i.e., it provides a file system that can be mounted and be accessed through the NFS protocol) that stores data encrypted. Application programs include `cmkdir`, which creates a CFS protected directory, `cattach`, which prepares a CFS directory for use, and `cdetach`, which reverses `cattach`'s operation.

Readers interested in details of CFS are advised to consult Blaze's paper [4].

2.2 Key Management

The goals of key management are as follows:

- A file key is derived from a master key in a smartcard.

  Only the owner of the smartcard should be able to use the file system. Therefore, a *file key*, which is used to encrypt and decrypt a file, should be derived from the master key in a smartcard. On the other hand, the master key should NOT be derivable from the file key.

- A unique file key is used to encrypt each file.

  A file key is used to encrypt only one file to minimize the damage if it is revealed through host compromise. This property is discussed more in Section 3.

- A file key changes with the associated file.

  When a file is written, its associated file key changes to protect the new file content; this provides forward secrecy. Consider the following scenario: a file key is stolen through host compromise. The file content is revealed to the adversary. Later, the file content is updated by a user. If the file key does not change, the new content is also available to the adversary. To avoid this, the file key should change on every update.

To achieve these goals, we designed the following key management scheme.

- A randomly generated master key is stored in a smartcard.
- `cfsd` uses a file's inode number and a timestamp of last modification as a seed of the file. Each entry is 4 bytes long, so the seed is 8 bytes long ({inode#, timestamp}).
- `cfsd` sends the seed to the smartcard. The card replies with the SHA1 hash result of the seed concatenated with the master key. (SHA1{inode#, timestamp, K_{user}}). This is 20 bytes long.
- `cfsd` further hashes the result into an 56-bit DES key, and uses it to encrypt and decrypt the file.

2.3 Authentication

SC-CFS employs the same authentication mechanism as CFS. A "signature", which is a 4 byte predefined string concatenated with 4 byte random string, encrypted in a way described in the previous key management section, is stored in each CFS directory. When a user starts accessing the directory, `cattach` tries to decrypt the signature. If `cattach` recovers the predefined string correctly, the user has entered the right password (in CFS) or used the right smartcard (in SC-CFS), so he is allowed to enter the directory.

In SC-CFS, before a smartcard is used, the correct *Personal Identification Number (or PIN)* must be typed. The PIN is a 3 - 8 digit number, which protects the information in the smartcard when it is lost or is stolen. The adversary who owns the smartcard cannot use it without knowing the PIN, as the smartcard blocks after some fixed number, say three, wrong PINs are entered.

2.4 Caching

CFS employs partial encryption of a file to minimize the performance overhead introduced by encryption. When a block (8 byte) in a file is updated, it is first XOR'ed with a precomputed string, encrypted with a sub key, and then XOR'ed with another precomputed string. The two precomputed strings and the sub key are pseudorandomly generated, based on the directory key [3]. The advantage of this approach over a chaining mode encryption, such as DES-CBC, is that a file can be partially updated. Chaining mode encryption requires the entire file to be encrypted at once.

As one of our goals is to change a key with every update of the associated file, we do not use this partial encryption approach. Instead, every write re-encrypts the whole file, . Therefore, DES-CBC is used.[4] This introduces potentially prohibitive performance overhead because of paging. In most UNIX systems, a file consists of several 4096 byte pages. A write operation to a long file is split into multiple 4096 byte writes. For example, to write a 1 Mbyte file, 256 write operations are necessary. We cannot afford to change a file key and encrypt the entire file 256 times. To counter this, a single file cache is introduced.

The cache loads a file when it is first accessed, and decrypts it. When the file is closed, it is encrypted under an updated file key and written back to the backing store. Because NFS does not have a `close` operation (NFS server is stateless), writeback happens in one of the following events:

[3]A *directory key* is a key used to encrypt files in a directory. This is entered by a user.

[4]We still could have used partial encryption to achieve partial reads. The decision to use DES-CBC may be reconsidered in the future.

- Another file enters the cache.
- Once a minute.
- CFS directory is detached.

3 Security Consideration

We discuss the security of our approach here, mainly in comparison with CFS. Another cryptographic file system, *Transparent Cryptographic File System (or TCFS)*, has a key management system similar to CFS [5]. Discussion of CFS in this section also applies to TCFS.

3.1 Model

We start with constructing a model of our system. The model consists of the following participants:

Alice (A) A user who uses CFS or SC-CFS.

Host A host computer that runs CFS or SC-CFS.

Smartcard A smartcard that plays the key generation role in SC-CFS.

Backing Store A backing store for CFS or SC-CFS. This may be any file system, e.g., a local file system or a network file system.

Mallory (M) An adversary.

3.2 Threats

We make the following assumptions in our model.

1. Mallory can compromise a host.

   Mallory can exploit security holes of the host, or physically access the host and overwrite the system administrator's password. Mallory can read and modify any information on the host.

2. Mallory cannot substantially change the behavior of the host.

   By Assumption 1, Mallory is able to install a Trojan horse in the host, which, for example, steals decrypted files. However, we assume this attack is impossible because:

   - Maintaining Trojan horses is hard, as Alice can find them by looking at change of file contents and logs.

- It becomes much harder if Alice uses application integrity checker, such as Tripwire [16].
- It becomes even harder if Alice uses hardware based integrity checker, such as AEGIS [1] and sAEGIS [13].

3. Mallory cannot compromise the smartcard.

   Mallory can neither read nor modify any information in a smartcard. She cannot influence the behavior of a smartcard.

4. Cryptographic operations are strong.

   Our principal cipher is DES, which is assumed impossible to compromise in reasonable amount of time. This may not be a good assumption any more in the age of fast DES crackers [7]. DES should be replaced with triple-DES in the future.

   Also, our principal hash function, SHA1, is assumed to be collision free.

3.3 Attack

Key Theft

If a host is compromised (possible by Assumption 1), keys can be stolen in CFS and SC-CFS:

In CFS, the key that encrypts the current working directory is stolen. As a result, all the files in the directory are revealed. Unless the key is explicitly changed, all the files will be accessible by the adversary.

In SC-CFS, the key that encrypts the file currently in the SC-CFS cache is stolen. The rest of the files in the file system are safe. The master key is safe because it is in a smartcard (Assumption 3). When the file is updated, it is encrypted under a different key, so it becomes safe again.

SC-CFS is more secure than CFS because when a key is stolen, only one file can be decrypted by the key. This file is being used by an application, so it resides in the clear in memory, and is revealed on host compromise, anyway. In contrast, when a directory key is stolen in CFS, all the files in the directory tree, including the ones that are not opened, are revealed.

CFS takes this "key per directory tree" approach to avoid forcing a user to remember many passwords. In SC-CFS, a smartcard remembers a randomly generated master key, and generates file keys, eliminating this problem.

Storage Theft

Storage theft is sometimes more easily accomplished than host compromise, thus requires special attention.

In CFS, the keys are derived by user passwords, and are vulnerable to dictionary attack. An adversary who steals a hard disk can run off-line dictionary attack as follows:

- Pick a password.
- Generate a sub key and random strings, as CFS does.
- Apply reversed CFS encryption operation to an encrypted file.
- If this recovers a readable text, this is the right key. If it does not, pick another password and try again.

Many sophisticated password crackers are published (e.g., John the Ripper [22]), and can be used to implement such an attack.

In SC-CFS, the master key is a random number, so it is not vulnerable to dictionary attack. By Assumption 4, brute force attack on the master key is also impossible.

On-Line Attack

In both CFS and SC-CFS, user authentication is performed by `cattach`, with a password in CFS and with a PIN in SC-CFS. As a consequence, if Mallory compromises the host (possible by Assumption 1) while Alice is using CFS or SC-CFS, she is able to impersonate Alice.

This causes more serious damage to CFS than to SC-CFS because with CFS, Alice has no way knowing Mallory is accessing her files. With SC-CFS, one can take advantage of physical isolation of a smartcard to counter this problem. For example, if a LED box that indicates data transmission via a serial port is installed on Alice's computer, she knows when Mallory is accessing her files. Furthermore, a display on the smartcard reader that displays the name of the accessed file and a pad on the reader that asks for a PIN on accessing files are useful.

The problem of on-line attack is a potent threat to almost all smartcard based systems because current smartcards do not have a secure I/O path with users. This is an important problem. A smartcard reader with a built-in PIN pad can solve this problem partially, i.e., it prevents PIN theft. SPYRUS's Rosetta Personal Access Reader 2 is an example of such readers [26].

Virtual Memory Compromise

Niels Provos has pointed out that virtual memory backing store may contain critical secrets even though application programs delete them [23]. By reading a hard disk which is used as the backing store, Mallory is able to recover secrets.

In CFS, the user master key and the directory keys may be in virtual memory. In SC-CFS, the user master key is in a smartcard, so only the file keys are vulnerable.

4 Implementation

Host-side implementation was tested on Linux-2.2.12 and OpenBSD-2.7. NFS is a standard protocol, so this should run on almost any UNIX. Smartcard-side implementation is specific to Schlumberger Cyberflex Access smartcard. Because Cyberflex Access is a Java card, we refer to the smartcard-side program as "SC-CFS applet".

SC-CFS has been implemented as extension to CFS. The implementation is divided into the following parts: modification to `cfsd`, `cattach`, `cmkdir`, and implementation of the SC-CFS applet. Here we discuss each part.

- Modification to `cfsd`

  In CFS, `cfsd` stores {inode#, creation time} in a file called `.pvect_`*encrypted-filename*. SC-CFS uses the same file. [5] First, `cfsd` is modified to store a modification time instead of a creation time, as the modification time is used as a seed of a file key in SC-CFS.

  Then, the single-file cache described in Section 2.4 is implemented. Finally, `read` and `write` operations are modified to access data through the cache.

- Modification to `cattach`

  When `cattach` is invoked with `-p port-number` option, it asks for a PIN instead of a password and then sends it to `cfsd`. `cfsd` initializes the smartcard, sends the PIN to the smartcard, and then carries out the card authentication described in Section 2.3.

- Modification to `cdetach`

  When `cdetach` is invoked, `cfsd` cleans up the cache and terminates the connection with the smartcard.

- Modification to `cmkdir`

  When `cmkdir` is invoked with `-S` option, it creates a signature described in Section 2.3 in the newly created SC-CFS directory.

- Implementation of SC-CFS applet

  The master key is stored in a file in a smartcard called `''ke''`, or `0x6b65`. This file is configured so that it cannot be accessed without going through the applet. The applet reads this file only after the correct PIN is presented. Key generation is simple: the applet concatenates the 8 byte seed to the 16 byte master key, hashes it with SHA1, and returns the result to `cfsd`.

[5] CFS does this instead of using information in the vnode structure, as the information changes on undesirable occasions, e.g., when a file is backed up and is resumed, or its modification time is changed by `touch`.

5 Performance Evaluation

We have evaluated the performance of SC-CFS in comparison with CFS and a local file system (EXT2). First, the result of the Andrew Benchmark Test [10] is reported to show user response time. Then, we look into the details of SC-CFS's most expensive operation: smartcard access.

The result shows that SC-CFS is significantly slower than CFS when it accesses a smartcard to generates keys. Most of this penalty is due to the slow speed of a smartcard.

All the measurements have been carried out on Linux-2.2 with 400 MHz AMD K6 and on Cyberflex Access smartcard. All the numbers reported are in seconds, and are average of 5 trials.

5.1 Round Trip Time

The Andrew Benchmark (ABM), a standard file system benchmark test, is used to measure the overhead of SC-CFS. ABM has five phases: MakeDir (`mkdir`), Copy (`cp`), ScanDir (`ls -l`), ReadAll (`grep`), and Make (`cc`). Source code of C programs used in the Make phase is slightly modified from the original Andrew Benchmark to make the test runnable on Linux-2.2 [6]. The results are shown below. The numbers are in seconds.

	Local	CFS (sec)	SC-CFS (sec)
MakeDir	0	0.2	0.2
Copy	0.6	1.0	21.8
ScanDir	1.2	1.6	1.0
ReadAll	2.0	3.0	22.6
Make	5.0	7.8	29.6

SC-CFS works as efficiently as Local and CFS when it does not need to access a smartcard (MakeDir and ScanDir [7]). However, in the other cases (Copy, ReadAll, Make), SC-CFS is much slower.

This performance impact is clearly due to the slow speed of a smartcard. Key generation, the only service the smartcard provides, takes 0.31 second. The following table shows: (1) the number of accesses to a smartcard, (2) (1) $\times$ the average smartcard access time (0.31), and (3) the difference between the round trip time of SC-CFS and CFS. The second column and the third are very close, showing that the most of the performance overhead is for smartcard.

	#acc	#acc $\times$0.31(s)	SC-CFS$-$CFS(s)
MakeDir	0	0	0
Copy	70	21.7	20.8
ScanDir	0	0	-0.6
ReadAll	70	21.7	19.6
Make	75	23.3	21.8

5.2 Detailed Look

As smartcard access is seen to be the bottleneck of SC-CFS, this part deserves special attention. Detailed performance evaluation was carried out on Cyberflex Access, which communicates at 57.6 Kbps with the host.

SC-CFS's smartcard operation involves two APDUs [8]: One is `generate_key`, which sends an 8 byte seed to the smartcard and invokes the key generation method inside the smartcard. The other is `get_response`, which asks the smartcard to return the result of key generation. A smartcard standard ISO 7816-4 [12] defines the `T=0` communication protocol, which Cyberflex Access adopts, to be unidirectional, i.e., a smartcard can either send or receive data in one APDU. Therefore, in addition to `generate_key` APDU, `get_response` APDU is necessary. These two APDUs are sent to the smartcard consecutively.

The following table shows the breakdown of the two APDUs. "`generate_key` APDU overhead" is time spent for sending a seed to smartcard, invoking the method, and preparing a buffer for returned data. Because this cannot be broken down further, it is shown as one operation.

operation	time (sec)
Hash (SHA1) 24 byte into 20 byte	0.15
`generate_key` APDU overhead	0.10
Select root in file system	0.01
Select key file "ke" in file system	0.01
Read 16 byte from key file	0.01
`get_response` APDU (20 byte)	0.01
total	0.29

The cost boils down to two dominating operations: SHA1 hash function and `generate_key` APDU overhead. These two are necessary operations, and we cannot improve the performance of them without modifying the smartcard. This points out the necessity of smartcards that execute cryptographic operations faster, with lighter method invocation overhead.

6 Related Work

There exist several remotely keyed encryption algorithms. The remotely keyed encryption is a way to encrypt bulk data with a key in a smartcard, where only small amount of work is done on the smartcard, and the rest is done on a fast host. Our session key generation scheme is one of them. Other examples include RKEP by Matt Blaze [3], another one by Blaze [2], ReMaRK [19] and ARK [18] by Lucks, and one by Jakobsson et al. [15].

We have chosen our keying scheme because this is implemented the fastest on the smartcard we used (Cyberflex Access). It is implemented with one

[6]We added five global variables, removed two `getchar()`s, and changed options to `ar`. None of them should alter performance significantly.

[7]File attributes retrieved by `stat()` are not encrypted.

[8]An *APDU* is a command sent to a smartcard from a host. Readers interested in smartcard concepts are advised to consult a reference text [9].

smartcard call, in which a hash function is called once. RKEP requires one smartcard call with two encryption calls and one hash function call. The other Blaze requires one smartcard call with one encryption call (encryption is more expensive than hash function in Cyberflex Access). ReMaRK requires two smartcard calls with two encryption function calls and one hash function call. ARK requires two smartcard calls with two random permutation calls and two random function calls. Jakobsson requires one smartcard call with one encryption call.

Our keying scheme appears to protect a master key and generates good session keys. However, not being cryptographers, we could not prove our scheme to be as secure as the other schemes in this paper. Whether our scheme is best for SC-CFS or we should choose one of the the other schemes is under discussion.

7 Future Direction

7.1 Administration Tools

With the current SC-CFS prototype, a user has to manually update his master key and PIN via our smartcard communication tool called pay [24]. Automated tools to do this should be provided.

7.2 Performance Improvement

Clearly, performance overhead is a large obstacle against wide deployment of SC-CFS. 300 millisecond overhead per file is acceptable for some applications, for example, word processing, but is not for others, such as scanning a large number of e-mail messages for a string, or a query operation on a large database. Therefore, performance improvement is essential.

Unfortunately, as shown in Section 5.2, the overhead is dominated by individual operations in a smartcard, which we, as application developers, cannot change. We hope new smartcards or other similar devices will achieve much higher performance in the near future.

To improve the performance of SC-CFS with current smartcard technology, it is possible to compromise between "key per directory tree" approach (CFS) and "key per file" approach (SC-CFS). The former is more efficient, but the latter is more secure. Depending on the security and performance requirements of an environment, middle ground implementation may be useful, e.g., "key per *n* files" approach, or caching file keys.

8 Conclusion

We have developed SC-CFS, which improves the security of CFS by integrating a smartcard as a personal secure storage of a key.

The following three aspects highlight the value of this work.

Improvement to important software.

As introduced in Section 1, the increasing threat of physical attack demands a way to protect secrets in a computer. CFS (and all the other file systems that protect files through encryption) is a secure and seamless solution to this problem. This work improves CFS in two important properties: security and convenience. SC-CFS is more secure than CFS because (1) the master key is a random number instead of a password, (2) the user master key is not exposed to the host, and (3) a stolen file key can reveal only one file. It is also more convenient than CFS because all a user has to remember is a short PIN, rather than multiple long passwords.

Important application for smartcards.

We believe that widespread deployment of secure hardware is essential to the security of computer systems. Systems are as secure as the weakest link, and reliance on user passwords is often the weakest link. Granted, passwords can be made stronger by choosing good ones and by changing them often. However, widespread deployment of security-critical Internet services requires a human to maintain many passwords: computer login, file system authentication, newspaper homepages, e-commerce homepages, online banks, web portals, and so on. Realistically speaking, it is impossible for a human to maintain so many good (therefore hard to remember) passwords. She will end up using the same password for many services, or will write the passwords down somewhere. Besides, she does not want to type passwords all the time. Smartcards solve this problem nicely by securely storing keys. Therefore, we wish to contribute to the widespread deployment of smartcards, and this work is an important step toward the goal, as secure storage is an important and suitable application of a smartcard (authentication being another [14]).

Remark on smartcard performance

Performance evaluation in Section 5 shows how important a fast smartcard is. In recent years, smartcards have matured in terms of functionality and reliability. However, we have not seen significant performance improvement, even though microprocessors have sped up by 5 to 10 times.

9 Availability

SC-CFS has been tested on Linux 2.2 and OpenBSD 2.7. The source code of SC-CFS is available at CITI homepage:

`http://www.citi.umich.edu/projects/smartcard/sc-cfs.html`

Acknowledgment

We thank Niels Provos for suggesting a high speed key generation method. Peter Honeyman and Jim Rees have advised us through this project, and suggested a file caching idea.

This work was partially supported by a research grant from Schlumberger, Inc.

References

[1] William A. Arbaugh, David J. Farber, and Jonathan M. Smith. A secure and reliable bootstrap architecture. In *1997 IEEE Symposium on Security and Privacy*, Oakland, CA, May 1997.

[2] M. Blaze. Key management in an encrypting file system. In *Proceedings of the USENIX Summer 1994 Technical Conference*, pages 27–35, Boston, MA, USA, 6–10 1994.

[3] M. Blaze. High-bandwidth encryption with low-bandwdith smartcards, 1996.

[4] Matt Blaze. A cryptographic file system for UNIX. In *Proceedings of 1st ACM Conference on Computer and Communications Security*, pages 9–16, Fairfax, Virginia, November 1993. ftp://ftp.research.att.com/ dist/mab/cfs.ps.

[5] G. Cattaneo, G. Persiano, A. Del Sorbo, A. Celentano, A. Cozzolino, E. Mauriello, and R. Pisapia. Design and implementation of a transparent cryptographic file system for UNIX. Unpublished Technical Report. Dip. Informatica ed Appl, Universita di Salerno. Available via ftp in ftp://edugw.dia.unisa.it/pub/tcfs/docs/tcfs.ps.gz., July 1997.

[6] Dorothy Denning. *Cryptography and Data Security*. Addison-Wesley, 1983.

[7] Electronic Frontier Foundation. *Cracking DES - Secrets of Encryption Research, Wiretap Politics & Chip Design*. O'Reilly & Associates, Inc., 1 edition, 1998.

[8] Jr. Frederick P. Brooks. *The Mythical Man-Month : Essays on Software Engineering*. Addison-Wesley, 2 edition, July 1995.

[9] Scott B. Guthery and Timothy M. Jurgensen. *Smart Card Developer's Kit*. MacMillan Technical Publishing, Indianapolis, Indiana, December 1997.

[10] J.H. Howard, M.L. Kazar, S.G. Menees, D.A. Nichols, M. Satyanarayanan, R.N. Sidebotham, and M.J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51 – 81, Feb. 1988.

[11] John D. Howard. Security incidents on the internet. In *Proceedings of INET 98*. Internet Society, 1998. http://www.comms.uab.es/inet99/inet98/ 2d/2d_3.htm.

[12] The International Organization for Standardization and The International Electrotechnical Commission. *ISO/IEC 7816-4 : Information technology - Identification cards - Integrated circuit(s) cards with contacts*, 9 1995.

[13] Naomaru Itoi, William A Arbaugh, Samuela J Pollack, and Daniel M Reeves. Personal secure booting. Technical report, Center for Information Technology Integration, 2000. To appear in ACISP 2001, Australia. Technical Report at http://www.citi.umich.edu/ techreports/.

[14] Naomaru Itoi and Peter Honeyman. Smartcard integration with Kerberos V5. In *Proceedings of USENIX Workshop on Smartcard Technology*, Chicago, May 1999.

[15] M. Jakobsson, J. Stern, and M. Yung. Encrypt small, 1999.

[16] Gene H. Kim and Eugene H. Spafford. The design and implementation of tripwire: A file system integrity checker. Technical report, Purdue University, 1995. CSD-TR-93-071.

[17] Daniel V. Klein. Foiling the cracker: A survey of, and improvements to, password security. In *UNIX Security Workshop II*, pages 5 – 14. USENIX Association, 1990.

[18] S. Lucks. Accelerated remotely keyed encryption. *Lecture Notes in Computer Science*, 1636:112–123, 1999.

[19] Stefan Lucks. On the security of remotely keyed encryption. In *Fast Software Encryption*, pages 219–229, 1997.

[20] Christopher Marquis. Deep security flaws seen at state dept, May 2000. http://www.fas.org/ sgp/news/2000/05/ nyt051100.html.

[21] Sun Microsystems. Network filesystem specification. Network Working Group, Request For Comments 1094, March 1989.

[22] Open Wall Project. John the ripper. http://www.openwall.com/john/.

[23] Niels Provos. Encrypting virtual memory. In *Proceedings of 9th USENIX Security Symposium*, August 2000.

[24] Jim Rees. Iso 7816 library, 1997. http://www.citi.umich.edu / projects / sinciti / smartcard / sc7816.html.

[25] Dug Song. dsniff. http://www.monkey.org/ dugsong/dsniff/.

[26] Spyrus. http:// www.spyrus.com/.

Secure Distribution of Events in Content-Based Publish Subscribe Systems

Lukasz Opyrchal and Atul Prakash
Electrical Engineering and Computer Science Department
University of Michigan
Ann Arbor, MI 48109-2122
{lukasz,aprakash}@eecs.umich.edu

Abstract

Content-based publish-subscribe systems are an emerging paradigm for building a range of distributed applications. A specific problem in content-based systems is the secure distribution of events to clients subscribing to those events. In content-based systems, **every** event can potentially have a different set of interested subscribers. To provide confidentiality guarantee, we would like to encrypt messages so that only interested subscribers can read the message. In the worst case, for *n* clients, there can be 2^n subgroups, and each event can go to a potentially different subgroup. A major problem is managing subgroup keys so that the number of encryptions required per event can be kept low. We first show the difficulties in applying existing group key management techniques to addressing the problem. We then propose and compare a number of approaches to reduce the number of encryptions and to increase message throughput. We present analytical analysis of described algorithms as well as simulation results.

1 Introduction

Many of today's Internet applications require high scalability as well as strict security guarantees. This new breed of applications includes large wireless delivery services with thousands to millions of clients, inter-enterprise supply-chain management applications, financial applications, workflow applications, and network management.

Messaging technology has been introduced to create much more flexible and scalable distributed systems. An emerging paradigm of messaging technology is *publish-subscribe* [B93]. In such systems, customers (or subscribers) specify the type of content they want to receive via subscriptions. Publishers publish messages (events), and the publish-subscribe system delivers them only to the interested subscribers. Publishers are often decoupled from subscribers, creating more scalable solutions. Figure 1shows a typical publish-subscribe system. Events may be delivered via intermediate *brokers*, who determine the set of subscribers that an event should be delivered to. Decoupling of publishers and subscribers works well for increasing scalability but, as we will see, makes it difficult to develop secure solutions.

This work is supported in part by the IBM Research Partnership Award and by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-00-2-0508. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, the Air Force Research Laboratory, or the U.S. Government.

The earliest publish-subscribe systems used subject-based subscription [B93, TIBCO]. In such systems, every message is labeled by the publisher as belonging to one of a fixed set of subjects (also known as groups, channels, or topics). Subscribers subscribe to all the messages within a particular subject or set of subjects. Strength of this approach is the potential to easily leverage group-based multicast techniques to provide scalability and performance, by assigning each subject to a multicast group. In fact, group communication can be considered to be a special case of subject-based subscription where the subject is the name of the group. A significant restriction with subject-based publish-subscribe is that the selectivity of subscriptions is limited to the predefined subjects.

An emerging alternative to subject-based systems is *content-based messaging systems* [BCM99, C98, CDF, GKP99, KR95, MS97, SA97]. These systems support an *event schema* defining the type of information contained in each event (message). For example, applications interested in stock trades may use the event schema `[issue: string, price: dollar, volume: integer]`. A content-based subscription is a *predicate* against the event schema, such as `(issue = "IBM" & price < 120 & volume > 1000)`. Only events that satisfy (match) the subscription predicate are delivered to the subscriber.

With content-based subscription, subscribers have the added flexibility of choosing filtering criteria along multiple dimensions, without requiring pre-definition of subjects. In the stock trading example, a subject-based subscriber could be forced to select trades by issue name because those are the only subjects available. In contrast, a content-based subscriber is free

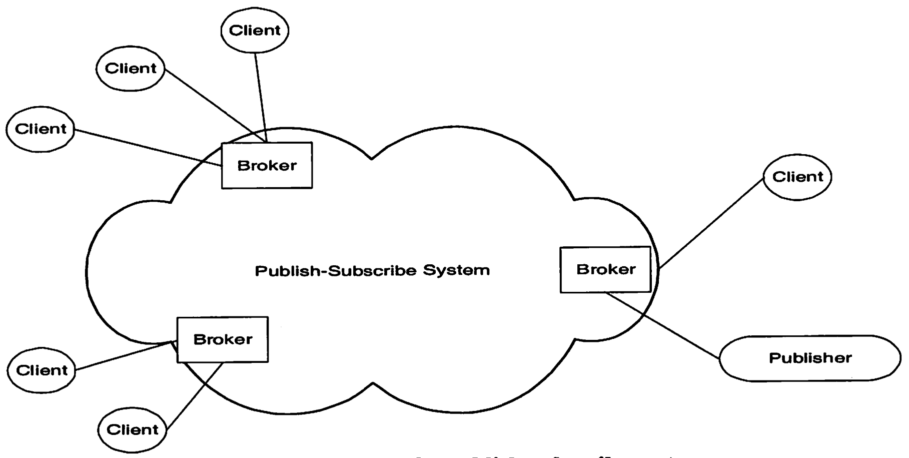


Fig. 1: An example of a publish-subscribe system.

to use any orthogonal criterion, such as volume, or indeed a predicate on any collection of criteria, such as issue, price, and volume.

The applications listed above require different security guarantees. For example, an application distributing premium stock reports may require *confidentiality* to make sure that only authorized (paying) subscribers can access the data. *Integrity* may be required to ensure that reports have not been modified in transit from publishers to subscribers and *sender authenticity* to make sure that fake reports are not sent by third parties. The lack of those security guarantees in content-based systems has prevented their wider use even in applications that could greatly benefit from content-based subscriptions.

The fact that in content-based systems every event can potentially go to a different subset of subscribers makes efficient implementation of confidentiality guarantees difficult. There are 2^N possible subsets, where N is the number of subscribers. With thousands (tens of thousands or hundreds of thousands) of subscribers it is infeasible to setup static security groups for every possible subset. Even the use of a limited number of intermediate trusted servers/brokers to reduce the complexity can leave each broker with hundreds or thousands of subscribers, making the number of possible groups too large.

This paper presents and compares several algorithms for secure delivery of events from a broker to its subscribers. Section 2 states the problem in detail. Section 3 presents related work. Section 4 explores a number of approaches based on the idea of using and caching multiple subgroup keys to address the secure end-point delivery problem. We described the use of these schemes and also present theoretical analysis of many of these approaches. Section 5 describes our simulation setup, experiment results and analysis of those results. Section 6 discusses the results and presents some theoretical bounds on the problem. Finally, Section 7 presents conclusions and directions for future work.

2 Problem Description

A messaging system routes events from a publisher to end-point brokers. The brokers then distribute those events to their subscribers. In content-based systems, each message could potentially go to a different set of subscribers (see Fig. 2). The picture shows two events (E_1 and E_2) delivered by the delivery system to the broker. Each event is then sent to a different subset of subscribers connected to the broker. We want to add certain security guarantees to content-based systems. The security requirement that we focus on in this paper is *confidentiality*. The system must guarantee that only authorized subscribers can read an event. Data must be protected from other (not authorized) subscribers as well as other malicious users on the network.

This paper describes issues and solutions for only a subset of the complex security problem in an entire publish-subscribe system. To provide event confidentiality, we assume that events are protected on their way from a publisher, through the delivery system, to the end-point brokers. In this paper, we focus on the data security on the last leg from end-point brokers to subscribers in an efficient way when each broker may have a large number of clients. In this paper, we assume that all brokers are trusted and that all subscribers and publishers are properly authenticated to the system. All subscribers and publishers also have an individual symmetric *pair-key* shared only with their broker (generated during the authentication process). The issues of security in transit between publishers and end-point brokers as well as the issue of broker trust are the subjects of our future work.

The publish-subscribe system allows dynamic access control to the events. This means that a predicate can be used at event publish time to check the set of subscribers who are authorized to receive the

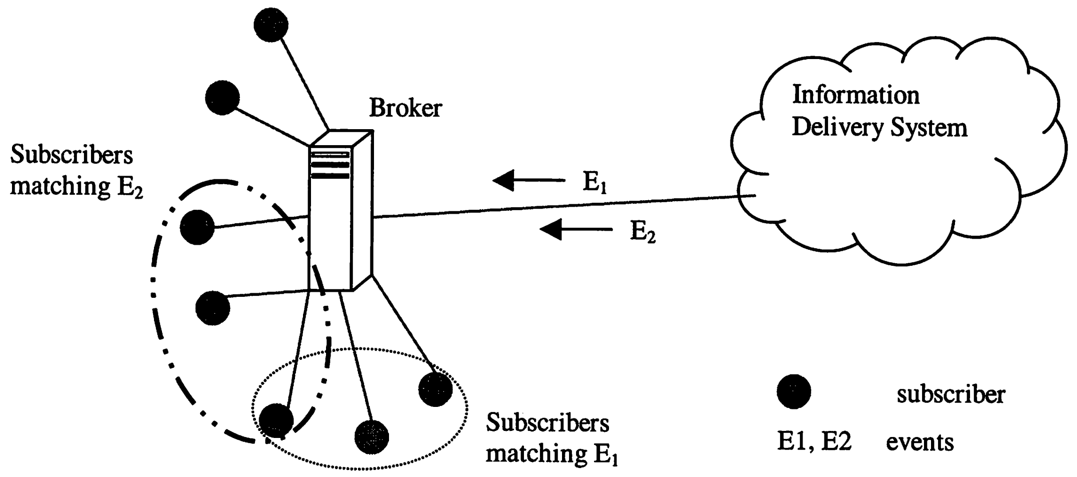


Fig. 2: Secure end-point delivery problem.

event. It is possible for a subscriber to be interested in an event (have a subscription for a particular event) but not be authorized to read that event (because of restrictions from a publisher). To simplify discussion, we assume that all interested subscribers are also authorized subscribers, i.e., each subscriber is authorized to read all events it subscribes to. Dynamic access control makes it infeasible to set up static security groups as each event can potentially have a different set of authorized subscribers (as well as a different set of interested subscribers).

Given the above restrictions and assumptions, providing confidentiality in content-based systems in an efficient way is non-trivial. Since every event can potentially go to a different subset of subscribers, it is infeasible (for large numbers of clients) to set-up static security groups. The simplest solution would be to encrypt each event separately for every subscriber receiving the event (using individual subscriber keys). For large systems where each broker has thousands of subscribers, this could mean hundreds or thousands of encryptions per event. An additional performance hit involves changing keys for each of the encryptions, which drastically slows down encryption algorithms like DES [Pub97]. We tested this by encrypting a random 64-bit piece of data with DES. We compared throughput when continuously encrypting with one key and when using 500 different keys on a Pentium III 550 Mhz machine running Red Hat Linux. The results showed that changing keys for each subscription results in throughput as low as 10% of the total throughput when using only one key.

In short, the problem presented here is to preserve confidentiality using small number of encryptions while distributing events from end-point broker to its subscribers. Due to lack of good workloads in this area, we consider two extreme scenarios --- (1) where events go to random subgroups of subscribers and (2) where there are some popular subgroups and some unpopular subgroups. The simplest solution is to encrypt each event separately for every subscriber receiving the event. This solution does not scale to large, high volume systems due to the throughput reduction of encryption algorithms like DES. This paper explores a number of dynamic caching approaches to reduce the number of encryptions and to increase message throughput.

3 Related Work

Until this point, the problem of efficiently delivering events in a confidential manner to only interested subscribers has not been addressed in content-based systems. However, a related and active area of research is secure group communication. Specifically, group key management services are closely related to the problem described above. Secure group communication systems are usually meant to provide secure channel for the exchange of data between group members. Secure groups are often identified by a *session key*, known to all group members, which is used to encrypt all data sent to the group. Key management services are used to facilitate member joins and leaves (including expulsions) as well as periodic re-keying to ensure validity of the session key.

A related area of work is research on *broadcast encryption*. It was first introduced by Fiat and Naor [FN93] in the context of pay-TV. The authors presented methods for securely broadcasting information such that only a selected subset of users can decrypt the information while coalitions of up to k unprivileged users learn nothing. Unfortunately, schemes presented in [FN93] as well as in extensions found in [BC94, BFMS98, SvT98] require a large numbers of keys to be stored at the receivers or large broadcast messages. Another problem in the context of secure content-based systems is that coalition of more than k unprivileged users can decrypt the information. Luby and Staddon [LS98] studied the trade-off between the number of keys stored in the receivers and the transmission length in large and small target receiver sets. They prove a lower bound that shows that either the transmission must be very long or a prohibitive number of keys must be stored in the receivers. An extension proposed in [ASW00] decreases the number of keys required and the length of transmissions by relaxing the target set. It allows a small fraction of users outside the target set to be able to decrypt the information. Such scheme may work well for pay-TV and similar applications but is unacceptable when confidentiality of broadcast information must be preserved.

The problem of secure event delivery from end-point brokers to subscribers can be cast as group communication with very dynamic membership. Since in content-based systems each event goes to a potentially different and arbitrary subset of subscribers, it is likely that two events arriving one after another go to completely different subsets of subscribers. If a group communication system were used to distribute events from a broker to subscribers, a group would have to be reconstructed (possibly entirely) for every event arriving at the broker. We describe a number of group key management approaches and show that none of the existing algorithms was designed to support dynamic membership changes that occur in content-based systems. The key management techniques for group communication are likely to have a large overhead when used in content-based system context.

A simple group key distribution method would be to create a *pair*-key between each subscriber and its broker. Whenever there is a membership change, the new session key is distributed to each member using its pair key. If an event requires a new subgroup of size N, this requires N encryptions to send a new session key. Since membership in content-based systems can potentially change for every event, this cost can be high for large groups.

One of the first attempts at standardizing secure multicast, GKMP [HM97a][HM97b] defines a protocol under which group session keys can be efficiently distributed. In GKMP, after being accepted into the group, newly joined members receive a Key Encrypting Key (KEK) under which all subsequent session keys are delivered. A limitation of this approach is that that there is no backward or forward secrecy. Anyone with the possession of the KEK can potentially access all the past and future session keys. The only way in GKMP to provide backward and forward secrecy is to reform the group with a new KEK. Obviously, the GKMP protocol does not support confidentiality requirements of content-based systems where events need to be protected from all but interested subscribers for that specific event.

Mittra's Iolus system [M97] attempts to overcome the problems in scalability of key distribution by introducing locally maintained subgroups. Each subgroup maintains its own session key, which is modified on membership events. Subgroups are arranged in a tree hierarchy. This solution is more scalable than the simple group key distribution method for membership changes. And, in fact, our approach of assigning subscribers to brokers is similar to the approach of subgrouping in Iolus. However, in the worst case, note that the session keys for all the subgroups may need to be changed for each event. And changing the key in a subgroup could potentially require linear number of encryptions in the number of subscribers within each subgroup.

The cost of changing keys in Iolus within subgroups can be reduced by making smaller subgroups. In the extreme case, for example, a small fixed number, K, of subscribers can be assigned to each subgroup, irrespective of the total number of subscribers, N. In that case, however, one ends up with a large number of intermediate servers (N/K) who must be trusted with the content of all the events sent via the system. We would like to explore solutions that reduce the number of servers we trust with event contents to as low values as feasible.

Approaches based on logical key hierarchies [WHA98][WGL98][YL00] provide an efficient approach to achieving scalable, secure key distribution on membership changes in group communication systems. A logical key hierarchy (LKH) is singly rooted *d-ary* tree of cryptographic keys (where d is usually 2, but can be arbitrary). A trusted session leader assigns the interior node keys, and each leaf node key is a secret key shared between the session leader and a single member. Once the group has been established, each member knows all the keys along the path from their leaf node key and the root. As changes in membership occur, re-keying is performed by replacing only those keys known (required) by the leaving (joining) member. It can be shown that the total cost of re-keying in key hierarchies in response to a single join or leave scales logarithmically with group size [WHA98][WGL98]. If the change in membership from the initial tree is $O(N)$, as is the case with a random group change, it may require up to $O(N \log(N))$ number of encryptions if members are removed (or added) individually. We consider that too high. If the tree is entirely reconstituted for each event being sent to a subgroup of size k, the number of encryptions required per event is at least k, which can still be large.

Another approach based on *LKH* uses the intermediate keys of the full tree. Instead of rebuilding the tree to represent the appropriate subgroup for each event, the tree can be searched to find the *smallest* subset of intermediate keys that cover all subscribers interested in the particular event. The search process is *O(N)* but the number of encryptions can be much smaller than in the tree rebuilding case. We compare our algorithms to this scheme in Section 5.

The VersaKey system [WCS99] extends the LKH algorithm by converting the key hierarchy into a table of keys, based on binary digits in the identifiers of the members. In this scheme, in the case of joins, no key distribution to current members is necessary. However, the VersaKey approach is vulnerable to collusion of ejected members. For example, two colluding members with complimentary identifiers cannot be ejected without simultaneously replacing the entire table. It is likely therefore that the table would need to be replaced for a large percentage of events going through the end-point broker.

A number of *key agreement* protocols (as opposed to *key distribution* approaches outlined above) have been suggested in which group keys are created from contributions of inputs (or key shares) of desired members [ITW82, SSD88, BD94, BW98, STW00, KPT00]. The general approach taken in these protocols is to extend the 2-party Diffie-Helman exchange protocol to *N* parties. Due to the contributory nature and perfect key independence, most of these protocols require exponentiations linear in the number of members [Steiner 2000] for most group updates. Kim et al unify the notion from key hierarchies and Diffie-Helman key exchange to achieve a cost of *O(log(N))* exponentiations for individual joins; however, the number of exponentiations for *k* joins or leaves will still be usually at least linear in *k* (unless the nodes that are leaving happen to be all clustered in the same parts of the key tree). Since exponentiation is expensive, these protocols are primarily suitable for establishing small groups at present and not suitable when group membership can change drastically from event to event. We thus exclude them for further consideration in this paper, and instead focus on approaches based on key distribution protocols.

Since the secure end-point delivery problem is only a part of a larger publish-subscribe system, we briefly describe related publish subscribe systems as well.

Relatively few event distribution systems [W98] allow subscriptions to be expressed as predicates over the entire message content. A few noteworthy examples of this emerging category are SIENA [C98], READY [GKP99], Elvin [SA97], JEDI [CDF], Yeast [KR95], GEM [MS97], and Gryphon [BCM99]. All of these systems support rich subscription predicates, and thus face problems of scalability in their event distribution algorithms. None of the above systems offers any security features.

Other, traditionally subject-based, publish-subscribe systems are also moving towards richer subscription languages. The Java Message Service (JMS) [SUN] enables the use of *message selectors*, which are predicates over a set of message *properties*. The OMG Notification Service [OMG] describes structured events with a “filterable body” portion. The TIB/Rendezvous system [TIBCO] available from the TIBCO Corporation has a hierarchy of subjects and permits subscription patterns over the resulting segmented subject field, also approximating some of the richness available with content-based subscription.

A new version of the Elvin system, Elvin 4 [SAB00], introduces a notion of keys for security [ABH00]; the details available are sketchy but it appears that the correct use and distribution of keys is up to the clients and servers, rather than being automatically managed by the underlying infrastructure based on subscriptions and event contents.

4 Group Key Caching

Our main goal is to reduce the number of encryptions required to ensure confidentiality when sending events from end-point brokers to subscribers. This reduction in number of encryptions in turn increases event throughput necessary for large and scalable systems. Due to the complexity of the problem – there are 2^N possible subgroups (where *N* is the number of subscribers) and every event can potentially go to a different subgroup – this paper explores only dynamic caching algorithms.

Our algorithms are compared to the naïve solution (described below) and the number of encryptions required by the naïve solution is our upper bound. Our algorithms must use less encryptions than the naïve algorithm without introducing other performance overheads. The naïve approach is described in figure 3:

1. new event E arrives at a broker and is matched to a set of subscribers
   G = [S1, S2, …, SL] (where S1, …, SL indicate interested and authorized subscribers)
2. generate a new key K_G
3. create a message $[\{E\}_{K_G}, \{K_G\}_{K_{S1}}, \{K_G\}_{K_{S2}}, \dots, \{K_G\}_{K_{SL}}]$ and send it to subscribers S1 through SL

Fig. 3: Naïve approach.

For every event arriving at an end-point broker and matching to K subscribers, the naïve approach needs K encryptions. For a broker with N subscribers, and a random distribution of groups (sets of subscribers interested in an event), an average event goes to N/2 subscribers. This means that with 1000 subscribers per broker, a broker must on average perform 500 encryptions per event.

Our approaches aim to improve on that number. All of our dynamic caching algorithms require the broker and subscriber to keep a certain size cache. In general, the cache stores keys for most popular groups. Cache entries have the following format: $< G, K_G >$

G is a bit vector identifying which subscribers belong to the group and K_G is the key associated with the group. K_G is used to encrypt events going to this particular set of subscribers. We assume that all subscribers have enough resources to cache all necessary keys. Subscriber S_1 must cache every entry $< G_X, K_X >$ cached at the broker such that $S_1 \in G$. In practice, subscribers with limited resources may have smaller caches. This means that such subscriber may not have all the appropriate keys cached at the broker. A secure protocol for key request/exchange has to be developed to in order for our caching algorithms to support subscribers with limited resources. Currently, we assume every subscriber has all the cache entries $\{< G, K > \mid S \in G\}$ cached at the broker.

The next few sections describe each of our algorithms in detail. We present theoretical analysis and simulation results comparing each of our approaches and the naïve solution. We also present simulation results for *LKH*-based solution for comparison. We derive approximate expressions for the average number of encryptions for two different distributions of groups:

Random – each arriving event goes to a random subset of subscribers. Every one of the possible 2^N groups has the same probability of occurrence.

Popular Set[1] – there is a set of groups that happen more often than others. An event has a higher probability of matching a group from the *popular group set* S. The distribution has the following parameters:

|S|: the size of set *S* (number of groups in the *popular set*)

p: probability that an event matches a group from S

Every event matches a random group from S with probability *p*. Every event matches a totally random group with probability (p – 1).

[1] The groups are based on subscriptions and therefore it is virtually impossible that every one of the 2^N possible groups has the same probability of occurrence. The popular set distribution is meant to better approximate real distribution.

To enable simpler derivation, we assume the cache to be *smart*. This means that it caches only groups from set S (if cache size is less than or equal to |S|). In practice, it is possible to closely approximate this behavior by using a *frequency-based* cache. By caching groups that occur more frequently, this approach would cache groups from set S after long enough time (since groups from S happen more frequently than other random groups).

We introduce a measure of average number of encryptions per message *E*. Due to uniform distribution of subscribers in a group, the average number of subscribers in a group is N/2, so:

$$E_{naive} = \frac{N}{2} \qquad \textbf{(1)}$$

4.1 Simple Caching

The simplest solution to reducing the number of encryptions in the system is to use a plain caching scheme. This approach assumes that, based on customer subscriptions, many events will go to the same subset of subscribers. Creating and caching a separate key for those groups would take advantage of *repeating groups* and it would reduce the number of encryptions performed at the broker. The major parameters affecting the performance of this approach are the number of clients, cache size, and the distribution of groups. The basic algorithm works as follows:

```
1. new event E arrives at a broker and is matched to a
   set of subscribers G = [S1 … SN]
2. search the current cache
   2.1. if an entry <G, K_G> is found in cache
        • send {E}_KG to all subscribers in G
   2.2. if entry <G, K_G> is not found in cache
        • generate a new key K_G
        • create the following message and send
          it to subscribers:
          [{E}_KG, {K_G}_KS1, …, {K_G}_KSN ]
          and send it to the set of subscribers G
        • add new entry <G, K_G> to cache
```

This approach works well if many events need to be delivered to the exact same set of subscribers. We present approximate formulas for average numbers of encryptions:

Random distribution: we know that if there is a cache hit (incoming event matches a group stored in cache), the event only needs to be encrypted once with the stored key. If there is a cache miss, then the number

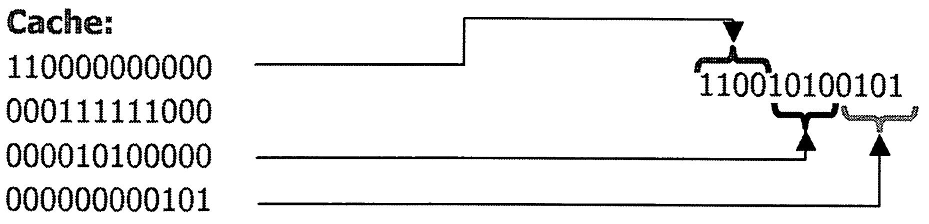


Fig. 4: Build-up cache.

of encryptions is the same as in the naïve case. The probability of a cache hit is:

$$P_{ch} = \frac{C}{2^N} \quad \text{(2a)}$$

Where C is the cache size and N is the number of subscribers. The probability of a cache miss is:

$$P_{cm} = \frac{2^N - C}{2^N} \quad \text{(2b)}$$

From equations 2a and 2b we can get the expression for the average number of encryptions per message:

$$E = \left[\left(\frac{C}{2^N} * 1\right) + \left(\frac{2^N - C}{2^N} * \frac{N}{2}\right)\right] \quad C < 2^N \quad \textbf{(2)}$$

We can see that as the number of subscribers gets larger, the simple cache will perform only slightly better than the naïve approach. For 100 subscribers and a cache of size 10,000, this approach averages 50 encryptions per message (just like the naïve solution).

Popular-set distribution: there are 2 cases: whether the group comes from the *popular set* or not (since the *smart* cache only caches groups from the *popular set*). If the group does not come from the *popular set* (with probability (1-*p*)) then the average number of encryptions is:

$$E_{np} = \frac{N}{2} \quad \text{(3a)}$$

If the group comes from the popular set, then the number of encryptions depends on whether there was a cache hit or not. Similarly to the simple cache with random groups, the average number of encryptions for groups coming from the *popular set* is:

$$E_p = \left[\left(\frac{C}{|S|} \times 1\right) + \left(\frac{|S| - C}{|S|} \times \frac{N}{2}\right)\right] \quad C < |S| \quad \text{(3b)}$$

Combining equations 4a and 4b, we get the expression for average number of encryptions with simple cache and *popular set* distribution:

$$E = (1-p)\frac{N}{2} + p\left[\left(\frac{C}{|S|} \times 1\right) + \left(\frac{|S| - C}{|S|} \times \frac{N}{2}\right)\right]$$

$$C < |S| \quad \textbf{(3)}$$

For the same parameters as in the random distribution case and the popular set of size 10,000 and probability $p = .9$, simple cache requires only 6 encryptions on average.

4.2 Build-up Cache

This algorithm extends the simple caching approach. The base of this solution is an observation that many groups are subsets of other, larger groups. In other words, larger groups can often be constructed by combining smaller groups. The algorithm takes advantage of this observation by performing *partial cache matching*. It searches for subsets of a given group G trying to reconstruct G using a number of smaller groups already in the cache. Figure 4 shows the basic idea behind the algorithm. Optimally, the algorithm would try to find a minimum number of groups from cache that completely *cover* group G. This would make the algorithm intractable. We use a heuristic that reduces the complexity of the search process to O(C), where C is the size of the cache. We search the cache from largest to smallest entries based on the assumption that finding a larger matching group will lead to a better overall match. The details are as follows:

1. new event E arrives at a broker and is matched to a set of subscribers G
2. search the cache for subgroups of G, starting from largest size groups
   - 2.1. if a full match is found (entry < G, K_G > is found in cache)
     - send $\{E\}_{KG}$ to all subscribers in G
   - 2.2. if a partial match is found (< G1, K_{G1} > where $G1 \subset G$ and $G1 \neq G$)
     - store < G1, K_{G1} > in a temporary set S
     - take the difference of $G_{new} = G - G1$
3. Repeat 2 until $G_{new} = \varnothing$ or there are no more entries in cache to search
4. if a match for G is not found in cache
   - save new entry in cache as in step 2.2 of the simple algorithm

Due to complexity, we cannot present a derivation of a formula for the average number of encryptions for the *build-up* cache at this time. We compare this approach to other algorithms in a set of simulations presented in section 5.

4.3 Clustered Cache

A known technique for reducing complexity of certain problems is clustering [reference]. There are 2^N possible groups and the above approaches may require very large cache sizes when the number of subscribers grows into thousands. This section describes a new clustered cache technique, which needs much smaller number of encryptions than the first two algorithms.

For a server with N clients, we divide the set of clients into K clusters. The server has to keep K separate *cluster caches*, but those caches can be much smaller than the caches from section 4.1 and 4.2. Each cluster cache holds entries of the same format as the simple and buildup caches ($< G, K_G >$), but G only consists of subscribers belonging to the particular cluster. The algorithm works in the following way:

```
1. new event E arrives at a broker and is matched
   to a set of subscribers G
2. G is divided into K subsets according to the
   cluster choices ($G_1, G_2, \ldots, G_K$)
3. for each cluster
   3.1. search cluster cache for the appropriate
        group (one of $G_1$ through $G_K$)
        • the algorithm works as the simple
          cache approach for each cluster
        • send message to the appropriate
          group in the cluster
        • add new entries to cluster cache as
          dictated by the simple algorithm
```

An event has to be encrypted separately for each cluster. Assuming a cache hit in every cluster, an event has to be encrypted K times.

An interesting issue in this algorithm is the choice of clusters. A simple solution is to assign subscribers to clusters randomly (or according to subscriber ids: first X subscribers to cluster 1, next X to cluster 2, etc.). Another approach would be to assign clusters based on subscription similarity. Subscribers with similar subscriptions would be assigned to the same cluster.

We calculate the average number of encryptions per each cluster and multiply it by the number of clusters to get the total. A cluster of size N_K, with individual cache of size C_k is very much like the simple cache from previous section. We present approximate formulas for *random distribution* only. Derived from equation 2, the average number of encryptions per cluster is:

$$E_K = \left[\left(\frac{C_K}{2^{N_K}} * 1\right) + \left(\frac{2^{N_K} - C_K}{2^{N_K}} * \frac{N_K}{2}\right)\right] \quad (4a)$$

The average number of encryptions for clustered cache is $E_k * K$, so:

$$E = \left[\left(\frac{C_K}{2^{N_K}} * 1\right) + \left(\frac{2^{N_K} - C_K}{2^{N_K}} * \frac{N_K}{2}\right)\right] * K$$

$$C_K \le 2^{N_K} \quad \textbf{(4)}$$

To simplify derivation and the resulting formulas we do not account for the fact that some clusters may have no subscribers matching a particular event, therefore reducing the number of encryptions. Equation 4 gives us an upper bound on the average number of encryptions for the clustered cache and random group distribution. For the same parameters as in the previous algorithms, the clustered cache requires at most 11 encryptions on average.

Next section describes an algorithm combining clustered and simple caches that combines the advantages of both approaches.

4.4 Clustered-Popular Cache

The clustered cache works well in a generic case of random groups. It doesn't take advantage of the fact that some groups occur more frequently, as in the *popular-set* distribution. The *clustered-popular* algorithm was designed to enhance clustered cache with support for frequently occurring (popular) groups. The basic idea is to combine clustered cache with simple cache in one. For each event, both caches are checked. If there is no hit in the simple part of the cache, the clustered approach is used to reduce the number of encryptions. The algorithm works as follows:

```
1. new event E arrives at a broker and is matched
   to a set of subscribers G
2. search the simple cache
   2.1. if an entry <G, $K_G$> is found in cache
        • send $\{E\}_{KG}$ to all subscribers in G
   2.2. if entry <G, $K_G$> is not found in cache
        • generate new key $K_G$ and add new
          entry <G, $K_G$> to cache
        • search the clustered cache as in
          section 4.3
        • send messages to the appropriate
          group in each cluster
```

We derive formulas for average number of encryptions for both *random* and *popular-set* distributions of groups. The formulas are based on the appropriate formulas for simple and clustered cache

$$E = \left(\frac{C}{2^N}\right)*1+\left(\frac{2^N-C}{2^N}\right)*\left[\left(\frac{C_K}{2^{N_K}}*1\right)+\left(\frac{2^{N_K}-C_K}{2^{N_K}}*\frac{N_K}{2}\right)\right]*K \quad (5)$$

$$E_{NP} = \left[\left(\frac{C_K}{2^{N_K}}*1\right)+\left(\frac{2^{N_K}-C_K}{2^{N_K}}*\frac{N_K}{2}\right)\right]*K \quad (6a)$$

$$E_P = \frac{C}{|S|}*1+\frac{|S|-C}{|S|}*\left[\frac{C_K}{2^{N_K}}*1+\frac{2^{N_K}-C_K}{2^{N_K}}*\frac{N_K}{2}\right]*K \quad (6d)$$

$$E = (1-p)\left[\left(\frac{C_K}{2^{N_K}}*1\right)+\left(\frac{2^{N_K}-C_K}{2^{N_K}}*\frac{N_K}{2}\right)\right]*K+ p\left[\frac{C}{|S|}*1+\frac{|S|-C}{|S|}*\left(\frac{C_K}{2^{N_K}}*1+\frac{2^{N_K}-C_K}{2^{N_K}}*\frac{N_K}{2}\right)*K\right] \quad (6)$$

Fig. 5: Formulas for numbers of encryptions.

approaches. We present the derivation separately for both group distributions.

Random distribution: if there is hit in the simple part of the cache, we need only 1 encryption. Otherwise, the number of encryptions is the same as in the clustered cache. Here, C is the size of the simple part of the cache and C_K is the size of clusters in the clustered part of the cache. We assume the following two conditions:

$$C_K \le 2^{N_K} \text{ and } C \le 2^N$$

With those conditions, the approximate formulas for the average number of encryptions are shown in figure 5, equation 5.

Popular distribution: We know that if the group does not come from the popular set (probability (1 – p)), there is no hit in the simple part of the cache (smart cache and we assume that C < |S|). In this case, the average number of encryptions is based only on the clustered part of the cache (figure 5, equation 6a).

If the group comes from the popular set then there is a hit in the simple cache with probability

$$P_{PS} = \frac{C}{|S|} \quad (6b)$$

In case of simple cache miss, clustered cache is checked with probability

$$P_{PC} = \frac{|S|-C}{|S|} \quad (6c)$$

The average number of encryptions when a group comes form the popular set S is shown in figure 5, equation 6d

Combination of equations 6a and 6d gives us a formula for the average number of encryptions for the *clustered-popular* cache and *popular-set* group distribution (figure 5, equation 6).

Table 1 shows average numbers of encryptions calculated using formulas derived above for two different sets of parameters. The clustered-popular approach always uses half of the cache for the simple part and second half for the clustered part. The **small set** has the following parameters: there are 100 clients and cache size is 10,000 entries. The clustered approach uses 10 clusters of 10 subscribers each with cluster cache size of 1000. The clustered-popular cache uses 11 clusters of about 9 subscribers each. The *popular set* distribution uses a set of 10,000 groups and probability p = .9. The **large set** has the following parameters: there are 1,000 clients and cache size is 100,000 entries. The clustered cache uses 100 clusters of 10 subscribers. The clustered-popular cache uses 114 clusters of 8 or 9 subscribers each. The popular set has 100,000 groups and probability p = .8.

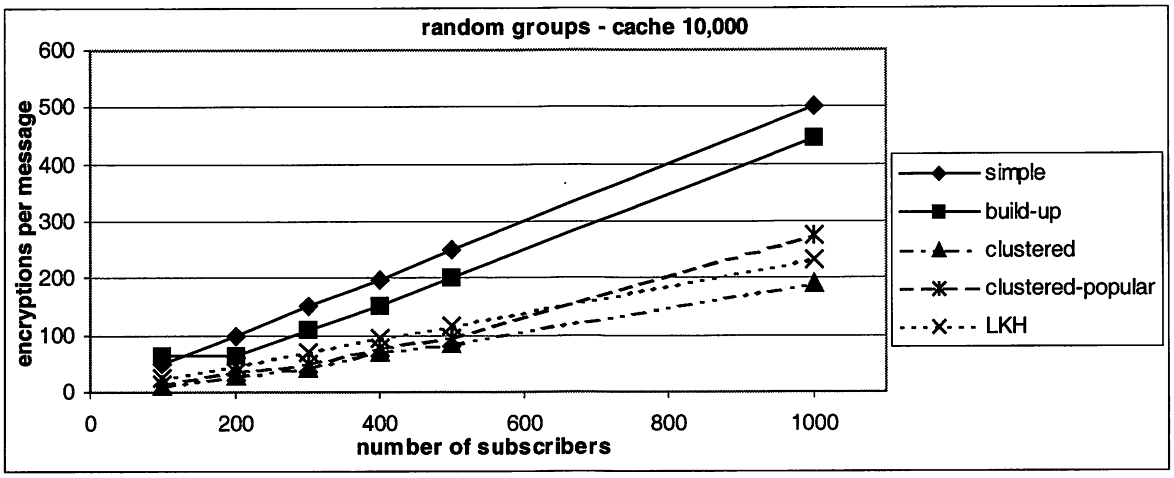
		No cache	Simple cache	Clustered cache	Clustered-popular cache
Small	Uniform groups	50	50	11	17
	Popular groups	50	6	~11	6.7
Large	Uniform groups	500	500	109	112
	Popular groups	500	101	~109	68

Table 1: Calculated average numbers of encryptions

Clustered-popular cache performs best in the large set and the popular-set group distribution. Next section shows that the clustered-popular approach outperforms other algorithms as the size of the cache increases. The actual numbers for both clustered algorithms should be lower than the theoretical values. This is because we did not account for the fact that for some events, not all clusters will have interested subscribers. This was done to simplify derivation. Also, the values for simple cache should be higher because of our assumption of *smart* cache. We were not able to approach the smart cache because of the size of our sample in the simulations.

5 Simulations

We ran a number of simulations to confirm our theoretical results as well as to compare the simple, clustered, and clustered-popular approaches to the build-up cache as well as to an LKH-based approach



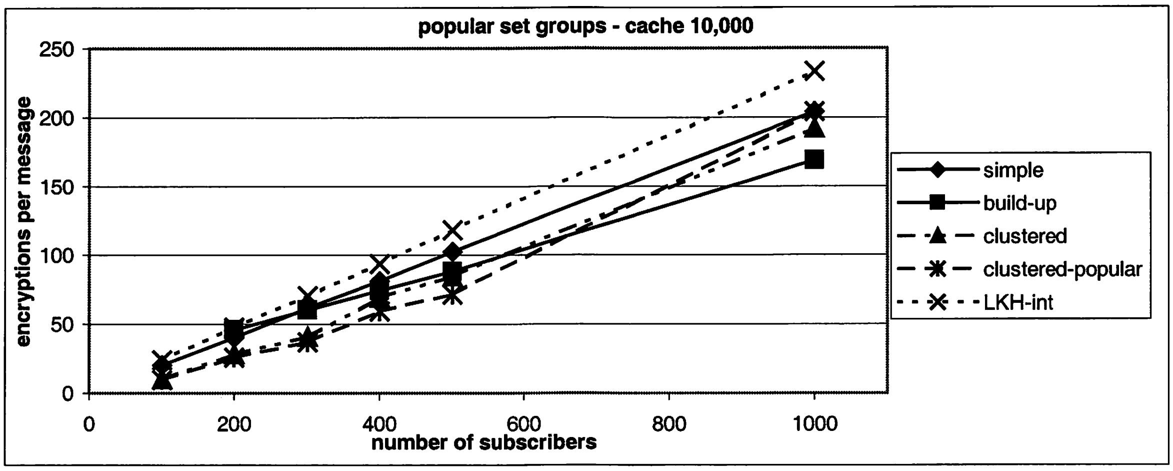


Fig. 6: Number of encryptions, cache size 10,000.

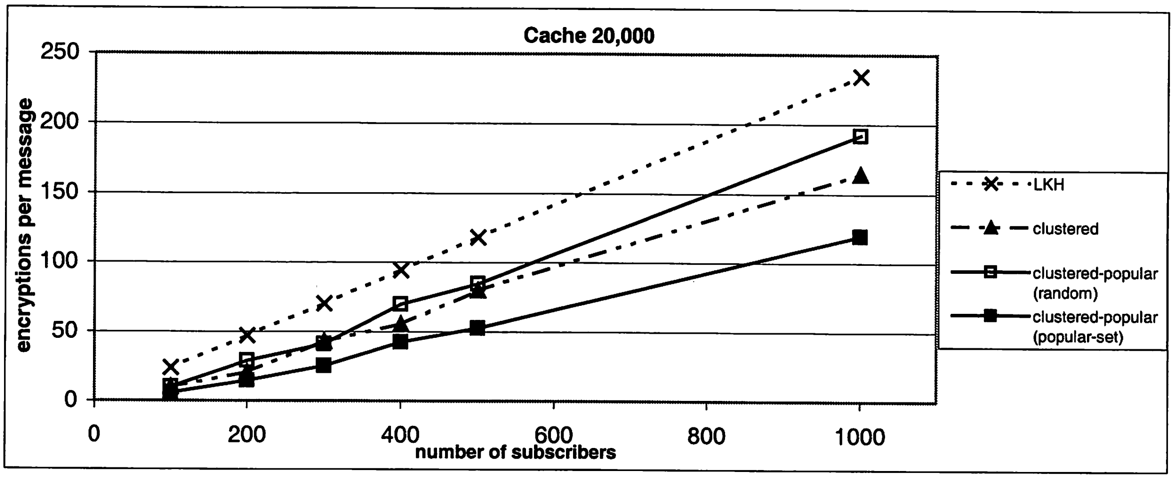


Fig. 7: Number of encryptions, cache size 20,000.

presented for comparison. The LKH based approach works by creating a hierarchical tree of keys. When an event arrives at broker and is matched to a group of subscribers G, the tree is searched, bottom-up, for keys that are common to as many as possible subscribers from G but are not know to subscribers not in G. The LKH-based scheme corresponds to a fixed cache size – the key tree only changes with changes in the number of subscribers. We ran a number of simulations testing different parameter settings. Each simulation consisted of 15,000 events. All caches were *warm*. We show results for both, *random* and *popular-set*, group distributions. The number of clusters used by the cluster-based algorithms depends on the number of clients. When solving equation 4 for the number of clusters K, we get the optimal number of clusters when $2^{N_K} = C_K$ (N_K is the number of clients per clusters and C_K is the cluster cache size). We chose the largest K which gave $2^{N_K} \geq C_K$. We used the same method to determine number of clusters for the clustered popular approach.

Figure 6 shows results for cache size 10,000, and figure 7 shows results for cache size 20,000. Figure 7 shows results for only the clustered, clustered-popular, and LKH-based approaches.

The clustered-popular is the only algorithm that performs differently with popular-set group distribution. We can see that clustered, clustered-popular, and LKH-based algorithms outperform the simple and build-up caches for random groups. In the case of popular-set distribution, all perform similarly with LKH-based approach being the worst. When the cache size is increased to 20,000, the clustered approaches clearly outperform the LKH-based algorithm. Figure 8 shows the effect of increasing cache size on the number of encryptions required by each algorithm for different group distributions. The results (except for the LKH-based approach, which is based on simulation results) in this figure are based on the approximate formulas derived in section 4. As we can see, clustered and clustered-popular approaches are similar with random group distribution, but the clustered-popular algorithm clearly outperforms all other solutions when the popular-set distribution is used. In order to make judgments about usefulness of any of these algorithms, we need to relate number of encryptions to a performance measure like throughput. We claimed that algorithms that reduce the number of encryptions required are desirable because large number of encryptions per message reduces message throughput at the broker. We measured throughput of the DES algorithm depending on the number of encryptions per message to show that this is the case. We ran a number of experiments encrypting an 8-byte piece of data. We varied the number of different keys used. We used the DES algorithm on a 550Mhz Pentium III running RedHat Linux operating system.

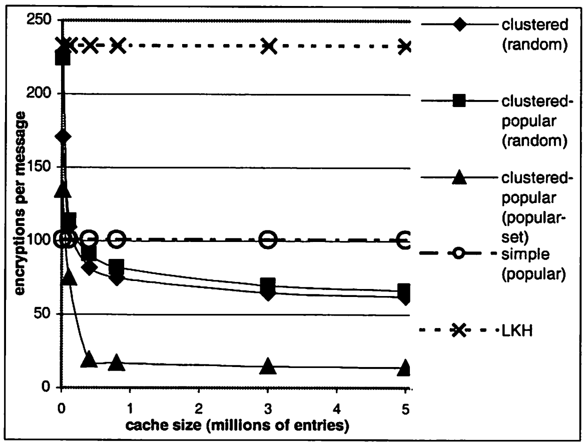


Fig. 8: Effects of cache size on the number of encryptions.

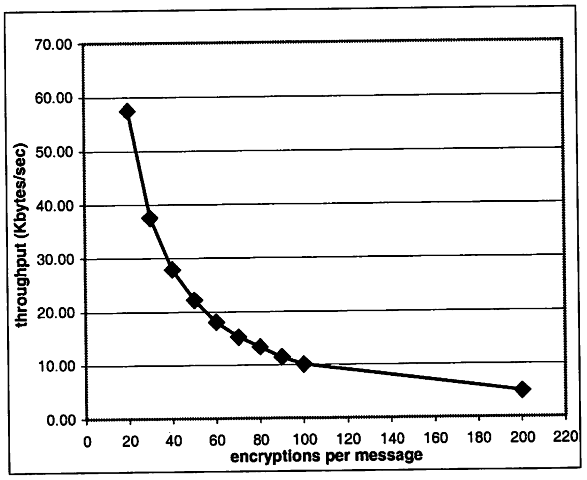


Fig. 9: Message throughput as a function of the number of encryptions per message.

The results of our experiment are showed in fig. 9. The adverse effect of number of encryptions on throughput is clearly visible. Based on our results, we see that clustered-popular has higher throughput than LKH-based approach by approximately 100% in the case of popular-set distribution and by about 33% in the case of random group distribution. Clustered-popular is also about 50% better than clustered algorithm for popular-set distribution, it, however, underperforms clustered algorithm by about 10-20% in the case of random distribution. The LKH-based approach also has approximately 40% lower throughput than the clustered algorithms.

6 Discussion

We can draw the following conclusions from the above results:

- clustering of users into an appropriate number of subgroups can substantially reduce the number of encryptions required for both the random case and the popular case.
- caching added to clustering can further reduce the number of encryptions substantially (with corresponding increase in message throughput).
- build-up cache, where previously cached keys are used to generate new keys, has remarkably little effect on number of encryptions required over simple cache, besides being more expensive to search when for entries that should be used to generate a new key.

The LKH scheme performed better than the simple cache for the random case despite the fact that the LKH scheme essentially requires a fixed amount of storage for a given number of clients (e.g., 1999 individual and internal group keys for 1000 clients), but not as well as clustering schemes. LKH scheme can also be more expensive to use in terms of search costs --- algorithmically, finding an optimal cover of LKH keys for a new subgroup usually takes more steps than searching the cluster-cache.

The reduction over simple caching and LKH that we considered appears to be asymptotically a constant factor for the same size cache. A question that arises is whether it is possible to do reduce the number of encryptions to *O(log N)* per event in the worst case, by either using multiple static LKH trees or by using a large cache. The answer appears to be no as the following analysis shows unless the cache size is exponential in the number of clients. Since the internal nodes of an LKH tree can also be considered to be cache entries, the result also says that an exponential number of LKH trees are required to reduce the worst-case bound on the number of encryptions to be *O(log(N))* per event. In the analysis below, we only consider the situation where generating a group key for a new event uses an optimal combination of existing keys in the cache.

Suppose the cache size is S. If we are allowed to pick at most p entries from the cache to form any subgroup, the maximum number of subgroups that can be formed from this cache is bounded by:

$$S^p$$

But, there are 2^N subgroups that need to be formed, given N clients. Therefore, if each potential subgroup needs to be formed using at most p cache entries, the following must hold:

$$S^p \geq 2^N$$

Therefore,

$$p \log_2 S \geq N$$

Or $$S \geq 2^{\frac{N}{p}}$$

The above result tells us that if p is any sublinear function of *N* (e.g., *O(ln N)* or *O(√N)*), the size of the cache must grow exponentially in *N* to guarantee that each event can be sent securely using at most p encryptions.

So, it appears that the worst-case number of encryptions for an event will need to be linear in *N* for reasonably sized caches, but one can try to improve the constant of proportionality, as we have tried to do in this paper. Note that this result applies only if a new event needs to be encrypted individually using existing (potentially multiple) group keys. An open question is whether one can do better if one allows an event to be encrypted multiple times; e.g., send a message encrypted under both keys k1 *and* k2, to send the message to only those members who possess both k1 and k2. Another open question is whether sublinear amortized bounds can be achieved without exponential-size caches. We leave the analysis of these questions to future work.

7 Conclusion and Future Work

There is a growing need for security solutions for content-based systems. This paper identifies the "secure end-point delivery" problem and explores a number of possible solutions. We are concerned with providing confidentiality when sending events from brokers to subscribers. The problem is that in content-based systems, **every** event can potentially have a different set of interested subscribers. There are 2^N possible subsets, where N is the number of subscribers. With thousands of subscribers it is infeasible to setup static security groups for every possible subset.

A number of key management systems for group communication solve a similar problem but none of them was designed to handle the dynamic nature of content-based event delivery. We explored a number of dynamic caching approaches. A simple solution is to encrypt each event separately for each interested subscriber; however this requires a large number of encryptions for large sets of subscribers. Our main goal is to reduce the number of encryptions required to preserve confidentiality while sending events only to interested subscribers. The number of encryptions is important because it translates directly into message throughput (see figure 9).

All of our approaches use a dynamic caching scheme, where the broker and subscribers must cache subgroup keys. Each cache entry has a format < G, K >, where G identifies the set of subscribers belonging to a subgroup and K is a key associated with the subgroup. All secure communication intended for all subscribers in G can be encrypted using key K. Through theoretical analysis and simulation results, we show that our clustered and clustered-popular approaches perform better as cache size increases. Both cluster-based algorithms outperform LKH-based solution for most cases. Clustered-popular algorithm performs especially well in the case of the *popular-set* group distribution. We also show that it is impossible to achieve sublinear encryptions growth for a large class of algorithms, even with using multiple LKH trees, without an exponential number of LKH trees or exponential-sized caches in the number of subscribers per broker.

Our results show that cluster-based algorithms can be a practical solution to the end-point delivery problem. They do not impose heavy overhead as hash tables can be used for cache lookup. The cache size requirement on the subscriber side is also lower than the simple cache or build-up cache algorithms. In the case of the clustered cache, each subscriber only needs C_k cache entries, where $C_K = \frac{C}{K}$. C is the total cache size at the broker, K is the number of clusters and C_K is the size of one cluster part of cache. If client resources are limited, a protocol for key request/exchange is needed for a subscriber to request appropriate keys from the broker.

We plan to investigate the effect of providing *integrity* and *sender authentication* on message throughput in the future. Efficient sender authentication in the context of content-based systems is a non-trivial problem. Throughout the paper we make an assumption that brokers are trusted. Every broker in the system has the ability to read every event. We are investigating the impact of non-universal broker trust on the design of algorithms.

References

[ABH00] D. Arnold, J. Boot, M. Henderson, T. Phelps, and B. Segall. Elvin – Content-Addressed Messaging Client Protocol. *Internet Draft, Network Working Group*, 2000. Available from http://elvin.dstc.edu.au/download/internet-draft.txt.

[ASW00] M. Abdalla, Y. Shavitt, and A. Wool. Key Management for Restricted Multicast Using Broadcast Encryption. *IEEE/ACM Transactions on Networking*, 8(4), pp 443-454, August 2000.

[B93] K. Birman. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM*, 36(12):37 53, December 1993.

[B96] A. Ballardie. Scalable Multicast Key Distribution. *Internet Engineering Task Force*, May 1996. RFC 1949.

[BC94] C. Blundo and A. Cresti. Space requirements for broadcast encryption. In *Advances in Cryptology – EUROCRYPT'94, LNCS 950*, pp 287-298. Springer-Verlag, 1994.

[BCM99] Guruduth Banavar, Tushar Chandra, Bodhi Mukherjee, Jay Nagarajarao, Robert E. Strom, and Daniel C. Sturman. An Efficient Multicast Protocol for Content-Based Publish-Subscribe Systems. In *International Conference on Distributed Computing Systems (ICDCS '99)*, June 1999.

[BD94] M. Burmester and Y. Desmedt. *A secure and efficient conference key distribution system. In Advances in Cryptology – EUROCRYPT '94*, 1995.

[BFMS98] C. Blundo, L. A. Frota Mattos, and D. R. Stinson. Generalized Beimel-Chor schemes for broadcast encryption and interactive key distribution. In *Theoretical Computer Science*, 200(1-2), pp 313-334, 1998.

[BW98] K. Becker and U. Wille. Communication complexity of group key distribution. In *5th ACM Conference on Computer and Communications Security*, San Francisco, CA, November 1998.

[C98] Antonio Carzaniga. *Architectures for an Event Notification Service Scalable to Wide-Area Networks*. PhD thesis, Politecnico di Milano, December 1998. Available: http://www.cs.colorado.edu/~carzanig/papers.

[CDF] G. Cugola, E. DiNitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. Submitted to Transactions on Software Engineering

[CDZ97] K. Calvert, M. Doar, and E. Zegura. Modeling Internet Topology. *IEEE Communications Magazine*, June, 1997.

[D89] S. Deering. *Host Extensions for IP Multicasting*. IETF RFC 1112, August 1989.

[GKP99] R. Gruber, B. Krishnamurthy, and E. Panagos. An Architecture of the READY Event Notification System. In *Proceedings of the Middleware Workshop at the International Conference on Distributed Computing Systems 1999, Austin, TX,* June 1999.

[HH99] Hugh Harney and Eric Harder. Group Secure Association Key Management Protocol (Draft). *Internet Engineering Task Force*, April 1999. draft-harney-sparta-gsakmp-sec-00.txt.

[HM97a] H. Harney and C. Muckenhirn. Group Key Management Protocol (GKMP) Architecture. *Internet Engineering Task Force*, July 1997. RFC 2094.

[HM97b] H. Harney and C. Muckenhirn. Group Key Management Protocol (GKMP) Specification. *Internet Engineering Task Force*, July 1997. RFC 2093.

[ITW82] I. Ingemarsson, D. Tang, and C. Wong. A conference key distribution system. *IEEE Transactions on Information Theory*, vol. 28, no. 5, pp. 714-720, September 1982.

[IONA] IONA Corporation. *OrbixTalk Fact Sheet*. http://www.iona.com/products/messaging/talk/index.html.

[KPT00] Y. Kim, A. Perrig, and G. Tsudik. Simple and Fault-Tolerant Key Agreement for Dynamic Collaborative Groups. In *Proceedings of 7th ACM Conference on Computer and Communication Security CCS 2000*.

[KR95] B. Krishnamurthy and D. Rosenblum. Yeast: A general purpose event-action system. *IEEE Transactions on Software Engineering*, 21(10), October 1995.

[LS98] M. Luby and J. Staddon. Combinatorial bounds for broadcast encryption. In *Advances in Cryptology – EUROCRYPT'98, LNCS 1403*, pp 512-526, Espoo, Finland, 1998.

[M97] S. Mittra. Iolus: A Framework for Scalable Secure Multicasting. In *Proceedings of ACM SIGCOMM '97*, pages 277 - 278. ACM, September 1997.

[MPH99] P. McDaniel, A. Prakash, and P. Honeyman. Antigone: A Flexible Framework for Secure Group Communication. In *Proceedings of the 8th USENIX Security Symposium*, August 1999.

[MS97] M. Mansouri-Samani and M. Sloman. A Generalised Event Monitoring Language for Distributed Systems. *EE/IOP/BCS Distributed Systems Engineering Journal*, 4(2), June 1997.

[MS98] David A. McGrew and Alan T. Sherman. Key Establishment in Large Dynamic Groups Using One-Way Function Trees. TIS Report No. 0755, TIS Labs at Network Associates, Inc., Glenwood, MD, May 1998.

[OAA00] L. Opyrchal, M. Astley, J. Auerbach, G. Banavar, R. Strom, and D. Sturman. Exploiting IP Multicast in Content-Based Publish-Subscribe Systems. In *Proceedings of Middleware 2000*, New York, April 2000.

[OMG] Object Management Group. *Notification Service*. http://www.omg.org/cgi-bin/doc?telecom/98-06-15

[P99] Adrian Perrig. Efficient Collaborative Key Management Protocols for Secure Autonomous Group Communication. In *International Workshop on Cryptographic Techniques and E-Commerce (CrypTEC '99)*, 1999.

[Pub77] Federal Information Processing Standards Publication. Data Encryption Standard, 1997. National Bureau of Standards.

[R94] M. Reiter. Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart. In *Proceedings of 2nd ACM Conference on Computer and Communications Security*, pages 68 - 80. ACM, November 1994.

[SA97] Bill Segall and David Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. *In Proceedings of AUUG97*, Brisbane, Australia, September 1997.

[SAB00] B. Segall, D. Arnold, J. Boot, M. Henderson, and T. Phelps. Content Based Routing with Elvin4. To appear in *Proceedings AUUG2K*, Canberra, Australia, June 2000.

[SSD88] D. Steer, L. Strawczynski, W. Diffie, and M. Wiener. A secure audio teleconference system. In *Advances in Cryptology – CRYPTO '88*, Santa Barbara, CA, August 1998.

[SvT98] D. R. Stinson and T. van Trung. Some new results on key distribution patterns and broadcast encryption. *Designs, Codes and Cryptography*, 14(3), pp261-279, 1998.

[STW00] M. Steiner, G. Tsudik, and M. Waidner. Key Agreement in Dynamic Peer Groups. *IEEE Transactions on Parallel and Distributed Systems*, 2000.

[SUN] Sun Microsystems. *Java Message Service*. http://java.sun.com/products/jms.

[TIBCO] TIBCO. *TIB/Rendezvous White Paper*. http://www.rv.tibco.com/whitepaper.html.

[VBM96] R. Van Renesse, K. Birman, and S. Maffeis. Horus: A Flexible Group Communication System. *Communications of the ACM*, 39(4):76 - 83, April 1996.

[WGL98] C. K. Wong, M. Gouda, and S. S. Lam. Secure Group Communication Using Key Graphs. In *Proceedings of ACM SIGCOMM '98*, pages 68 - 79. ACM, September 1998.

[WHA98] Debby M. Wallner, Eric J. Harder, and Ryan C. Agee. Key Management for Multicast: Issues and Architectures (Draft). *Internet Engineering Task Force, September 1998*. draft-wallner-key-arch-01.txt.

[WISEN98] Workshop on Internet Scale Event Notification. See http://www.ics.uci.edu/IRUS/wisen/wisen98 for details.

[WCS99] M. Waldvogel, G. Caronni, D. Sun, N. Weiler, and B. Plattner. The VersaKey Framework: Versatile Group Key Management. *IEEE Journal on Selected Areas in Communications*, 17(9), September 1999.

[YL00] Y. Yang and S. Lam. A Secure Key Management Protocol Communication Lower Bound. *Technical Report TR2000-24*, The University of Texas at Austin, Austin, TX, September 2000.

[ZCB96] E. Zegura, K. Calvert, and S. Bhattacharjee. How to Model an Internetwork. In *Proceedings of IEEE Infocom '96*, San Francisco, CA, April 1996.

A Method for Fast Revocation of Public Key Certificates and Security Capabilities*

Dan Boneh† dabo@cs.stanford.edu

Xuhua Ding‡ xhding@isi.edu

Gene Tsudik‡ gts@ics.uci.edu

Chi Ming Wong† bc@cs.stanford.edu

Abstract

We present a new approach to fast certificate revocation centered around the concept of an on-line semi-trusted mediator (SEM). The use of a SEM in conjunction with a simple threshold variant of the RSA cryptosystem (mediated RSA) offers a number of practical advantages over current revocation techniques. Our approach simplifies validation of digital signatures and enables certificate revocation within legacy systems. It also provides immediate revocation of all security capabilities. This paper discusses both the architecture and implementation of our approach as well as performance and compatibility with the existing infrastructure. Our results show that threshold cryptography is practical for certificate revocation.

1 Introduction

We begin this paper with an example to illustrate the premise for this work. Consider an organization – industrial, government or military – where all employees (referred to as *users*) have certain authorities and authorizations. We assume that a modern Public Key Infrastructure (PKI) is available and all users have digital signature, as well as encryption, capabilities. In the course of performing routine everyday tasks users take advantage of secure applications such as email, file transfer, remote log-in and web browsing.

Now suppose that a trusted user (Alice) does something that warrants immediate revocation of her security privileges. For example, Alice might be fired, or she may suspect that her private key has been compromised. Ideally, immediately following revocation, Alice should be unable to perform any security operations and use any secure applications. Specifically, this means:

- Alice cannot read secure (private) email. This includes encrypted email that is already residing on Alice's email server. Although encrypted email may be basically delivered (to Alice's email server), she cannot decrypt it.
- Alice cannot generate valid digital signatures on any further messages. (However, signatures generated by Alice prior to revocation may need to remain valid.)
- Alice cannot authenticate herself to corporate servers.

In Section 7, we discuss current revocation techniques and demonstrate that the above requirements are impossible to satisfy with these techniques. Most importantly, current techniques do not provide immediate revocation.

1.1 The SEM architecture.

Our approach to immediate revocation of security capabilities is called the SEM architecture. It is easy to use and its presence is transparent to peer users (those that encrypt messages and verify signatures). The basic idea is as follows:

We introduce a new entity, referred to as a SEM *(SEcurity Mediator). A* SEM *is an online semi-trusted server. To sign or decrypt a message, Alice must first obtain a message-specific token from the* SEM. *Without this token Alice cannot use her private key.*[1] *To revoke Alice's ability to sign or de-*

*This work is supported by the Defense Advanced Project Agency (DARPA) under contract F30602-99-1-0530.

†Computer Science Department, Stanford University.

‡Department of Information and Computer Science, University of California, Irvine.

[1]The exact description of the token is in Section 2.

crypt, the security administrator instructs the SEM *to stop issuing tokens for Alice's public key. At that instant, Alice's signature and/or decryption capabilities are revoked. For scalability reasons, a* SEM *serves many users.*

We emphasize that the SEM architecture is transparent to peer users: with SEM's help, Alice can generate a standard RSA signature, and decrypt standard messages encrypted with her RSA public key. Without SEM's help, she cannot perform either of these operations. The SEM architecture is implemented using threshold RSA [3] as described in section 2.

To experiment with this architecture we implemented it using OpenSSL [12]. SEM is implemented as a daemon process running on a server. We describe our implementation, the protocols used to communicate with the SEM, and give performance results in Sections 5 and 6.

We also built a plug-in for the Eudora client enabling users to send signed email. All signatures are generated with SEM's help (see [15]). Consequently, signing capabilities can be easily revoked.

1.2 Decryption and signing in the SEM architecture

We now describe in more detail how decryption and signing is done in the SEM architecture:

– Decryption: suppose Alice wishes to decrypt an email message using her private key. Recall that encrypted email is composed of two parts: (1) a short header containing a message-key encrypted using Alice's public key, and (2) the body contains the email message encrypted using the message-key. To decrypt, Alice first sends the short header to her SEM. SEM responds with a short token. This token enables Alice to read her email. However, it contains no useful information to anyone but Alice. Hence, communication with the SEM does not have to be protected or authenticated. We note that interaction with the SEM is fully managed by Alice's email reader and does not require any intervention on Alice's part. This interaction does not use Alice's private key. If Alice wants to read her email offline, the interaction with the SEM takes places at the time Alice's email client downloads Alice's email from the email server.

– Signatures: suppose Alice wishes to sign a message using her private key. She sends a hash of the message to the SEM which, in turn, responds with a short token enabling Alice to generate the signature. As with decryption, this token contains no useful information to anyone but Alice; therefore, the interaction with the SEM is not encrypted or authenticated.

Note that all interaction with the SEM involves very short messages.

1.3 Other benefits of using a SEM

Our initial motivation for introducing a SEM is to enable immediate revocation of Alice's key. We point out that the SEM architecture provides two additional benefits over standard revocation techniques: (1) simplified signature validation, and (2) enabling revocation in legacy systems. These benefits apply when the following semantics for validating digital signatures are used:

Binding signature semantics: a digital signature is considered valid if the certificate associated with the signature was valid at the time the signature was issued.

A consequence of binding signature semantics is that all signatures issued prior to certificate revocation are valid. Binding semantics are natural in business contracts. For example, suppose Alice and Bob enter into a contract. They both sign the contract at time T. Bob begins to fulfill the contract and incurs certain costs in the process. Now, suppose at time $T' > T$, Alice revokes her own certificate. Is the contract valid at time T'? Using binding semantics, Alice is still bound to the contract since it was signed at time T when her certificate was still valid. In other words, Alice cannot nullify the contract by causing her own certificate to be revoked.

(We note that binding semantics are inappropriate in some scenarios. For example, if a certificate is obtained from a CA under false pretense, e.g., Alice masquerading as Bob, the CA should be allowed to declare at any time that **all** signatures ever issued under that certificate are invalid.)

Implementing binding signature semantics with existing revocation techniques is complicated, as discussed in Section 7. Whenever Bob verifies a signature generated by Alice, Bob must also verify that Alice's certificate was valid at the time the signature was issued. In fact, every verifier of Alice's signature must perform this certificate validation step. However, unless a trusted timestamping service is involved in generating all of Alice's signatures, Bob cannot trust the timestamp provided by Alice in her signatures.

Implementing binding semantics with the SEM architecture is trivial. To validate Alice's signature, a verifier need only verify the signature itself. There is no need to check the status of Alice's certificate.[2] Indeed, once Alice's certificate is revoked she can no longer generate valid signatures. Therefore, the mere existence of the signature implies that Alices's certificate was valid at the time the signature was issued.

The above discussion brings out two additional benefits of a SEM over existing revocation techniques, assuming binding semantics are sufficient.

– Simplified signature validation. Verifiers need not validate the signer's certificate. The existence of a (verifiable) signature is, in itself, a proof of signature's validity.

– Enabling revocation in legacy systems. Consider legacy systems doing signature verification. Often, such systems have no certificate validation capabilities. For example, old browsers (e.g., Netscape 3.0) verify server certificates without any means for checking certificate revocation status. In SEM architecture, certificate revocation is provided without any change to the verification process in these legacy systems. (The only aspect that needs changing is the signature generation process. However, we note that, often, only a few entities generate signatures, e.g., CAs and servers.)

2 Mediated RSA

We now describe in detail how the SEM interacts with users to generate tokens. The proposed SEM architecture is based on a variant of RSA which we call Mediated RSA (mRSA). The main idea in mRSA is to split each RSA private key into two parts using threshold RSA [3]. One part is given to a user while the other is given to a SEM. If the user and the SEM cooperate, they employ their respective half-keys in a way that is functionally equivalent to (and indistinguishable from) standard RSA. The fact that the private key is not held in its entirety by any one party is transparent to the outside world, i.e., to the those who use the corresponding public key. Also, knowledge of a half-key cannot be used to derive the entire private key. Therefore, neither the user nor the SEM can decrypt or sign a message without mutual consent. (A single SEM serves a multitude of users.)

[2] We are assuming here that revocation of Alice's key is equivalent to revocation of Alice's certificate. In general, however, Alice's certificate may encode many rights, not just the right to use her key(s). It is then possible to revoke only some of these rights while not revoking the entire certificate.

2.1 mRSA in detail

Public Key. As in RSA, each user (U_i) has a public key $EK_i = (n_i, e_i)$ where the modulus n_i is product of two large primes p_i and q_i and e_i is an integer relatively prime to $\phi(n_i)$.

Secret Key. As in RSA, there exists a corresponding secret key $DK_i = (n_i, d_i)$ where $d_i * e_i = 1 \pmod{\phi(n_i)}$. However, as mentioned above, no one has possession of d_i. Instead, d_i is effectively split into two parts d_i^u and d_i^{sem} which are held by the user U_i and a SEM, respectively. The relationship among them is:

$$d_i = d_i^{sem} + d_i^u \mod \phi(n)$$

mRSA Key Setup. Recall that, in RSA, each user generates its own modulus n_i and a public/secret key-pair. In mRSA, a trusted party (most likely, a CA) takes care of all key setup. In particular, it generates a distinct set: $\{p_i,\ q_i,\ e_i,\ \text{and}\ d_i, d_i^{sem}\}$ for each user. The first four are generated in the same manner as in standard RSA. The fifth value, d_i^{sem}, is a random integer in the interval $[1, n_i]$. The last value is set as: $d_i^u = d_i - d_i^{sem}$.

After CA computes the above values, d_i^{sem} is securely communicated to a SEM and d_i^u – to the user U_i. The details of this step are elaborated in Section 5.

mRSA Signatures. A user generates a signature on a message m as follows:

1. The user U_i first sends a hash of the message m to the appropriate SEM.
   - SEM checks that U_i is not revoked and, if so, computes a partial signature $PS_{sem} = m^{d_i^{sem}} (\bmod n_i)$ and replies with it to the user. This PS_{sem} is the token enabling signature generation.
   - concurrently, U_i computes $PS_u = m^{d_i^u} (\bmod n_i)$
2. U_i receives PS_{sem} and computes $m' = (PS_{sem} * PS_u)^{e_i} (\bmod n_i)$. If $m' = m$, the signature is set to: $(PS_{sem} * PS_u) = m^{d_i} \quad (\bmod n_i)$.

Note that in Step 2 the user U_i validates the response from the SEM. Signature verification is identical to that in standard RSA.

mRSA Encryption. The encryption process is identical to that in standard RSA. (In other words, ciphertext is computed as $c = m^{e_i} (\bmod n_i)$ where m is an appropriately padded plaintext, e.g., using OAEP.) Decryption, on the other hand, is very similar to signature generation above.

1. upon obtaining an encrypted message c, user U_i sends it to the appropriate SEM.
   - SEM checks that U_i is not revoked and, if so, computes a partial cleartext $PC_{sem} = c^{d_i^{sem}} (\bmod n_i)$ and replies to the user.
   - concurrently, U_i computes $PC_u = c^{d_i^u} (\bmod n_i)$.
2. U_i receives PC_{sem} and computes $c' = (PC_{sem} * PC_u)^{e_i} (\bmod n_i)$. If $c' = c$, the cleartext message is: $(PC_{sem} * PC_u) = c^{d_i^u}$.

2.2 Notable Features

As mentioned earlier, mRSA is only a slight modification of the RSA cryptosystem. However, at a higher, more systems level, mRSA affords some interesting features.

CA-based Key Generation. Recall that, in RSA, a private/public key-pair is typically generated by its intended owner. In mRSA the key-pair is typically generated by a CA, implying that the CA knows the private keys belonging to all users. In the global Internet this is clearly undesirable. However, in a medium-sized organization this "feature" provides key escrow. For example, if Alice is fired, the organization can still access her work-related files by obtaining her private key from the CA.

If key escrow is undesirable, it is easy to extend the system in a way that no entity ever knows Alice's private key (not even Alice or the CA). To do so, we can use a technique due to Boneh and Franklin [2] to generate an RSA key-pair so that the private key is shared by a number of parties since its creation (see also [4]). This technique has been implemented in [8]. It can be used to generate a shared RSA key between Alice and the SEM so that no one knows the full private key. Our initial implementation does not use this method. Instead, the CA does the full key setup.

Immediate Revocation. The notoriously difficult revocation problem is greatly simplified in mRSA. In order to revoke a user's public key, it suffices to notify that user's SEM. Each SEM merely maintains a list of revoked users which is consulted upon every service request. Our implementation uses standard X.509 Certificate Revocation Lists (CRL's) for this purpose.

Transparency. mRSA is completely transparent to those who encrypt data for mRSA users and those who verify signatures produced by mRSA users. To them, mRSA appears indistinguishable from standard RSA. Furthermore, mRSA certificates are identical to standard RSA certificates.

Coexistence. mRSA's built-in revocation approach can co-exist with the traditional, explicit revocation approaches. For example, a CRL- or a CRT-based scheme can be used in conjunction with mRSA in order to accommodate existing implementations that require verifiers (and encryptors) to perform certificate revocation checks.

CA Communication. CA remains an off-line entity. mRSA certificates, along with private half-keys are distributed to the user and SEM-s in an off-line manner. This follows the common certificate issuance and distribution paradigm. In fact, in our implementation (Section 5) there is no need for the CA and the SEM to ever communicate directly.

No Authentication. mRSA does not require any explicit authentication between a SEM and a user. Instead, a user implicitly authenticates a SEM by verifying its own signature (or encryption) as described in Section 2.1. In fact, signature and encryption verification steps assure the user of the integrity of the communication with the SEM.

3 Architecture

The overall architecture is made up of three components: CA, SEM, and user.

A single CA governs a (small) number of SEMs. Each SEM, in turn, serves many users. The assignment of users to SEMs is assumed to be handled off-line by a security administrator. A user may be served by multiple SEM's.

Our CA component is a simple add-on to the existing CA and is thus considered an off-line entity. For each user, the CA component takes care of generating an RSA public key, a corresponding RSA public key certificate and a pair of half-keys (one for the user and one for the SEM) which, when combined, form the RSA private key. The respective half-keys are then delivered, out-of-band, to the interested parties.

The user component consists of the client library that provides the mRSA sign and mRSA decrypt operations. (As mentioned earlier, the verify and encrypt operations are identical to standard RSA.) It also handles the installation of the user's credentials at the local host.

The SEM component is the critical part of the architecture. Since a single SEM serves many users, performance, fault-tolerance and physical security are of paramount concern. The SEM is basically a daemon process that processes requests from its constituent users. For each request, SEM consults its revocation list and refuses to help sign (or decrypt) for any revoked users. A SEM can be configured to operate in a stateful or stateless model. The former involves storing per user state (half-key and certificate) while, in the latter, no per user state is kept, however, some extra processing is incurred for each user request. The tradeoff is fairly clear: per user state and fast request handling versus no state and somewhat slower request handling.

We now describe the SEM architecture in more detail. A user's request is initially handled by the SEM controller where the packet format is checked. Next, the request is passed on to the client manager which performs a revocation check. If the requesting user is not revoked, the request is handled depending on the SEM state model. If the SEM is stateless, it expects to find the so-called SEM *bundle* in the request. This bundle, as discussed in more detail later, contains the mRSA half-key, d_i^{SEM}, encrypted (for the SEM, using its public key) and signed (by the CA). The bundle also contains the RSA public key certificate for the requesting user. Once the bundle is verified, the request is handled by either the $\text{mRSA}_{\text{sign}}$ or $\text{mRSA}_{\text{decrypt}}$ component. In case of the appropriate signature request, the optional timestamping service is invoked. If the SEM maintains user state, the bundle is expected only in the initial request. The same process as above is followed, however, the SEM's half-key and the user's certificate are stored locally. In subsequent user requests, the bundle (if present) is ignored and local state is used instead.

The administrator communicates with the SEM via the admin interface. The interface enables the administrator to manipulate the revocation list.

4 Security of the SEM architecture

We now briefly summarize the security features of mRSA and the SEM architecture.

First, consider an attacker trying to subvert a user (Alice). The attacker's goal is to decrypt a message sent to Alice or to forge Alice's signature on a certain message. Recall that the token sent back to Alice is $t = x^{d^{sem}} \bmod N$ for some value of x. The attacker sees both x and the token t. In fact, since there is no authentication of the user's request to the SEM, the attacker can obtain this t for any x of its choice. We claim that this information is of no use to an attacker. After all, d^{sem} is just a random number in $[1, n]$ independent of the rest of the attacker's view. More precisely, we argue that any attack possible with the SEM architecture is also possible when the user uses standard RSA. This statement can be proven using a simulation argument. In attacking standard RSA one can simulate the SEM (by picking a random integer d^{sem} in $[1, n]$) and thus use the attack on the SEM to mount an attack on standard

RSA. Furthermore, the attacker cannot masquerade as the SEM since Alice checks all responses from the SEM as described in Section 2.1.

Suppose the attacker is able to compromise the SEM and expose the secret key d^{sem}. This enables the attacker to "unrevoke" revoked, or block possible future revocation of currently valid, certificates. However, knowledge of d^{sem} does not enable the attacker to decrypt messages or sign messages on behalf of users. Nevertheless, it is desirable to protect the SEM's key. A standard approach is to distribute the key among a number of SEM servers using secret sharing. Furthermore, the key should never be reconstructed at a single location. To extract the SEM's key an attacker would need to break into multiple SEM servers. When using mRSA, it is possible to distribute the SEM's secret in this way using standard techniques from threshold cryptography [3].

Once Alice's key is revoked, she cannot decrypt or sign messages using her private key. To show this, we argue that, if Alice could sign or decrypt messages using only her share of private key, then RSA is insecure.

Finally, note that each user is given her own random RSA modulus n_i. This means that if a number of users are compromised (or a number of users collude) there is no danger to other users. The private keys of the compromised users will be exposed, but private keys of all other users will remain unaffected.

5 Implementation

We implemented the entire SEM architecture for the purposes of experimentation and validation. The reference implementation is publicly available. Following the architecture described earlier, the implementation is composed of three parts:

1. CA and Admin Utilities:
   includes certificate issuance and revocation interface.

2. SEM daemon:
   SEM architecture as described in Section 3

3. Client libraries:
   mRSA user functions accessible via an API.

Our reference implementation uses the popular OpenSSL [12] library as the low-level cryptographic platform. OpenSSL incorporates a multitude of cryptographic functions and large-number arithmetic primitives. In addition to being efficient and available on many common hardware and software platforms, OpenSSL adheres to the common PKCS standards and is in the public domain.

The SEM daemon and the CA/Admin utilities are implemented on Linux and Solaris while the client libraries are available on both Linux and Windows98 platforms.

In the initialization phase, CA utilities are used to set up the RSA public key-pair for each client (user). The set up process follows the description in Section 2. Once the mRSA parameters are generated, two structures are exported: 1) SEM *bundle*, which includes the SEM's half-key d_i^{SEM}, and 2) user *bundle*, which includes d_i^u and the entire server bundle. The format of both SEM and user bundles conforms to the PKCS#7 standard.

The server bundle is basically an RSA envelope signed by the CA and encrypted with the SEM's public key. The client bundle is a shared-key envelope also signed by the CA and encrypted with the user-supplied key which can be a password or a passphrase. (A user cannot be assumed to have a pre-existing public key.)

After issuance, the user bundle is distributed in an out-of-band manner to the appropriate user. Before attempting any mRSA transactions, the user must first decrypt and verify the bundle. A separate utility program is provided for this purpose. With it, the bundle is decrypted with the user-supplied key, the CA's signature is verified, and, finally, the user's new certificate and half-key are extracted and stored locally.

To sign or decrypt a message, the user starts with sending an mRSA request with the SEM bundle piggybacked. The SEM processes the request and the bundle contained therein as described in Section 3. (Recall that the SEM bundle is processed based on the state model of the particular SEM.) All mRSA packets have a common packet header; the payload format depends on the packet type. The packet header is defined in Figure 1.

```
0              8               16             24              32
+--------------+---------------+--------------+---------------+
| PROTOCOL     | PACKET_TYPE   |      DATA_LENGTH             |
+--------------+---------------+--------------+---------------+

PROTOCOL   :  protocol identifier. Set to MRSA(=1) in current code
PACKET_TYPE:  one of the following:
         1) REG_REQ_T : register request
         2) REG_RLY_T : register reply
         3) SIG_REQ_T : signature request
         4) SIG_RLY_T : signature reply
         5) DEC_REQ_T : decrypt request
         6) DEC_RLY_T : decrypt reply
```

Figure 1: mRSA Packet Header

5.1 Email client plug-in

To demonstrate the ease of using the SEM architecture we implemented a plug-in for the Eudora email reader [15]. When sending signed email the plug-in reads the user bundle described in the previous section. It obtains the SEM address from the bundle and then communicates with the SEM to sign the email. The resulting signed email can be verified using any S/MIME capable email client such as Microsoft Outlook. In other words, the email recipient is oblivious to the fact that a SEM is used to control the sender's signing capabilities.

Figure 2 shows a screen snap shot of trying to send signed email using a revoked key. In this example, the plug-in contacts the SEM and is told that the SEM will not supply the token for a revoked key. Consequently, the plug-in displays a message informing the user that the email cannot be signed.

6 Experimental Results

We conducted a number of experiments in order to evaluate the practicality of the proposed architecture and our implementation.

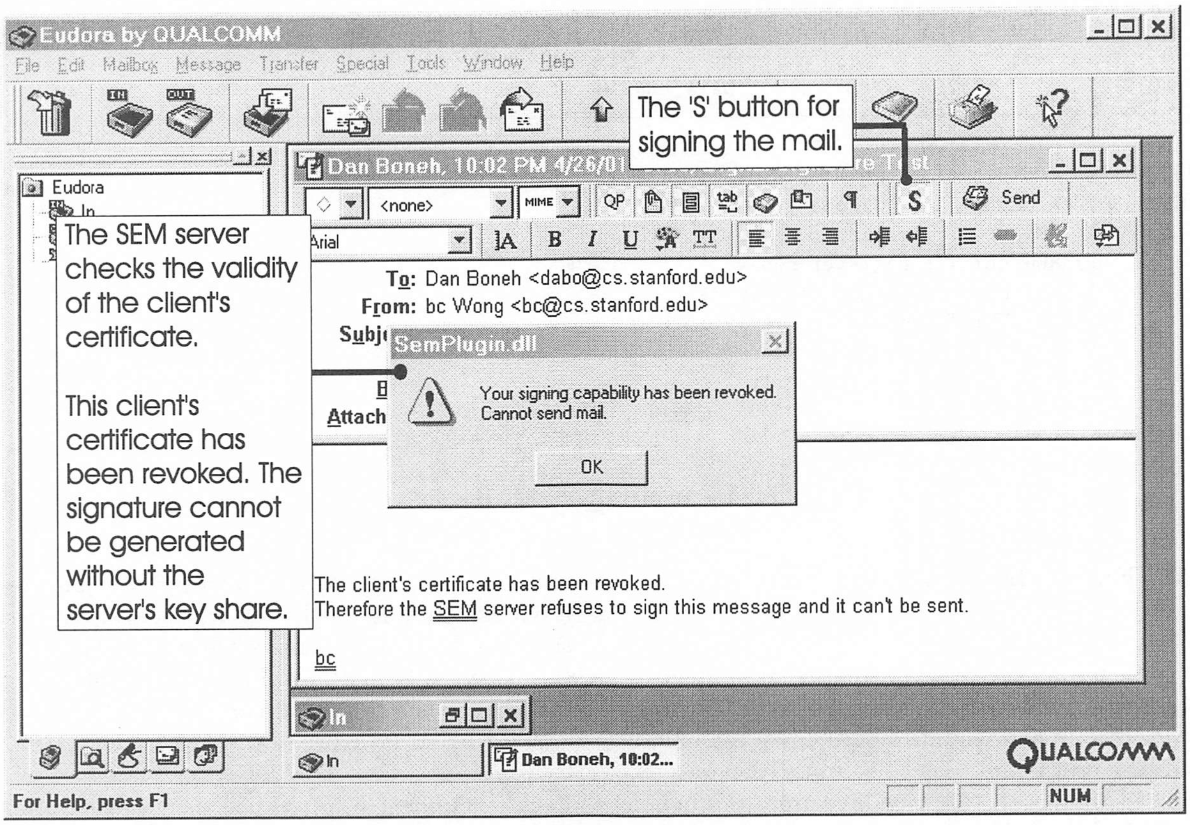
We ran the SEM daemon on a Linux PC equipped with an 800 Mhz Pentium III processor. Two different clients were used. The fast client was on another Linux PC with a 930 MHz Pentium III. Both SEM and fast client PC-s had 256M of RAM. The slow client was on a Linux PC with 466 MHz Pentium II and 128M of RAM. Although an 800 Mhz processor is not exactly state-of-the-art, we opted to err on the side of safety and assume a relatively conservative (i.e., slow) SEM platform. In practice, a SEM might reside on much faster hardware and is likely to be assisted by an RSA hardware acceleration card.

Each experiment involved one thousand iterations. All reported timings are in milliseconds (rounded to the nearest 0.1 *ms*). The SEM and client PCs were located in different sites interconnected by a high-speed regional network. All protocol messages are transmitted over UDP.

Client RSA key (modulus) sizes were varied among 512, 1024 and 2048 bits. (Though it is clear that 512 is not a realistic RSA key size any longer.) The timings are only for the mRSA sign operation since mRSA decrypt is operationally almost identical.

6.1 Communication Overhead

In order to gain precise understanding of our results, we first provide separate measurements for communication latency in mRSA. Recall that both mRSA operations involve a request from a client followed by a reply from a SEM. As mentioned above, the test PCs were connected by a high-speed regional network. We measured communication latency by varying the key size which directly influences message sizes. The results are shown in Table 1 (message sizes are in bytes). Latency is calculated as the round-trip delay between the client and the SEM. The numbers are identical for both client types.


The 'S' button for signing the mail.

The SEM server checks the validity of the client's certificate.

This client's certificate has been revoked. The signature cannot be generated without the server's key share.

To: Dan Boneh <dabo@cs.stanford.edu>
From: bc Wong <bc@cs.stanford.edu>

Your signing capability has been revoked.
Cannot send mail.

The client's certificate has been revoked.
Therefore the SEM server refuses to sign this message and it can't be sent.

bc

Eudora by QUALCOMM
File Edit Mailbox Message Transfer Special Tools Window Help
Dan Boneh, 10:02 PM 4/26/01
SemPlugin.dll
OK
Send
Eudora
In
For Help, press F1
NUM
QUALCOMM

Figure 2: Screen snapshot of SEM email plug-in

Keysize (bits)	Message Size (bytes)	Comm. latency (ms)
512	102	4.0
1024	167	4.5
2048	296	5.5

Table 1: Communication latency

6.2 Standard RSA

As a point of comparison, we initially timed the standard RSA sign operation in OpenSSL (Version 0.9.6) with three different key sizes on each of our three test PCs. The results are shown in Tables 2 and 3. Each timing includes a message hash computation followed by an exponentiation. Table 2 reflects optimized RSA computation where the *Chinese Remainder Theorem (CRT)* is used to speed up exponentiation (essentially exponentiation is done modulo the prime factors rather than modulo N). Table 3 reflects unoptimized RSA computation without the benefit of the CRT. Taking advantage of the CRT requires knowledge of the factors (p and q) of the modulus n. Recall that, in mRSA, neither the SEM nor the user know the factorization of the modulus, hence, with regard to its computation cost, mRSA is more akin to unoptimized RSA.

As evident from the two tables, the optimized RSA performs a factor of 3-3.5 faster for the 1024- and 2048-bit moduli than the unoptimized version. For 512-bit keys, the difference is slightly less marked.

6.3 mRSA Measurements

The mRSA results are obtained by measuring the time starting with the message hash computation by the user (client) and ending with the verification of the signature by the user. The measurements are

Keysize (bits)	466 Mhz PII (slow client)	800 Mhz PIII (SEM)	930 Mhz PIII (fast client)
512	2.9	1.4	1.4
1024	14.3	7.7	7.2
2048	85.7	49.4	42.8

Table 2: RSA results with CRT (in milliseconds).

Keysize (bits)	466 Mhz PII (slow client)	800 Mhz PIII (SEM)	930 Mhz PIII (fast client)
512	6.9	4.0	3.4
1024	43.1	24.8	21.2
2048	297.7	169.2	144.7

Table 3: Standard RSA results without CRT (in milliseconds).

illustrated in Table 4.

It comes as no surprise that the numbers for the slow client in Table 4 are very close to the unoptimized RSA measurements in Table 3. This is because the time for an mRSA operation is determined solely by the client for 1024- and 2048- bit keys. With a 512-bit key, the slow client is fast enough to compute its PS_u in $6.9ms$. This is still under $8.0ms$ (the sum of $4ms$ round-trip delay and $4ms$ RSA operation at the SEM).

The situation is very different with a fast client. Here, for all key sizes, the timing is determined by the sum of the round-trip client-SEM packet delay and the service time at the SEM. For instance, $178.3ms$ (clocked for 2048-bit keys) is very close to $174.7ms$ which is the sum of $5.5ms$ communication delay and $169.2ms$ unoptimized RSA operation at the SEM.

All of the above measurements were taken with the SEM operating in a stateful mode. In a stateless mode, SEM incurs further overhead due to the processing of the SEM bundle for each incoming request. This includes decryption of the bundle and verification of the CA's signature found inside. To get an idea of the mRSA overhead with a stateless SEM, we conclude the experiments with Table 5 showing the bundle processing overhead. Only 1024- and 2048-bit SEM key size was considered. (512-bit keys are certainly inappropriate for a SEM.) The CA key size was constant at 1024 bits.

7 Comparison of SEM with existing certificate revocation techniques

Certificate revocation is a well recognized problem with the existing Public Key Infrastructure (PKI). Several proposals address this problem. We briefly review these proposals and compare them to the SEM architecture. For each proposal we describe how it applies to signatures and to encryption. For simplicity we use signed and encrypted Email as an example application. We refer to the entity validating and revoking certificates as the Validation Authority (VA). Typically, the VA is the same entity as the Certificate Authority (CA). However, in some cases these are separate organizations.

A note on timestamping. Binding signature semantics (Section 1.3) for signature verification states that a signature is considered valid if the key used to generate the signature was valid at the time signature generation. Consequently, a verifier must establish exactly when a signature was generated. Hence, when signing a message, the signer must interact with a trusted timestamping service to obtain a trusted timestamp and a signature over the user's (signed) message. This proves to any verifier that a signature was generated at a specific time. All the techniques discussed below require a signature to contain a timestamp indicating when a signature was issued. We implicitly assume this service. As we will see, there is no need for a trusted time service to implement binding signature semantics with the SEM architecture.

Keysize (bits)	466 Mhz PII (slow client)	930 Mhz PIII (fast client)
512	8.0	9.9
1024	45.6	31.2
2048	335.6	178.3

Table 4: Timings for mRSA (in milliseconds).

SEM key size	Bundle overhead
1024	8.1
2048	50.3

Table 5: Bundle overhead in mRSA with a SEM in a stateless mode (in milliseconds).

7.1 Review of existing revocation techniques

CRLs and Δ-CRLs: Certificate Revocation Lists are the most common way to handle certificate revocation. The Validation Authority (VA) periodically posts a signed list of all revoked certificates. These lists are placed on designated servers called CRL distribution points. Since these lists can get quite long, the VA may alternatively post a signed Δ-CRL which only contains the list of revoked certificates since the last CRL was issued. For completeness, we briefly explain how CRLs are used in the context of signatures and encryption:

- Encryption: at the time email is sent, the sender checks that the receiver's certificate is not on the current CRL. The sender then sends encrypted email to the receiver.
- Signatures: when verifying a signature on a message, the verifier checks that, at the time that the signature was issued, the signer's certificate was not on the CRL.

OCSP: The Online Certificate Status Protocol (OCSP) [11] improves on CRLs by avoiding the transmission of long CRLs to every user and by providing more timely revocation information. To validate a specific certificate in OCSP, the user sends a certificate status request to the VA. The VA sends back a signed response indicating whether the specified certificate is currently revoked. OCSP is used as follows for Encryption and signatures:

- Signatures: When verifying a signature, the verifier sends an OCSP query to the VA to check if the corresponding certificate is currently valid. Note that the current OCSP protocol prevents one from implementing binding semantics: it is not possible to ask an OCSP responder whether a certificate was valid at some time in the past. Hopefully this will be corrected in future versions of the protocol.

  One could potentially abuse the OCSP protocol and provide binding semantics as follows. To sign a message, the signer generates the signature, and also sends an OCSP query to the VA. The VA responds with a signed message saying that the certificate is currently valid. The signer appends both the signature and the response from the VA to the message. To verify the signature, the verifier checks the VA's signature on the validation response. The response from the VA provides a proof that the signer's certificate is currently valid. This method reduces the load on the VA: it is not necessary to contact the VA every time a signature is verified. Unfortunately, there is currently no infrastructure to support this mechanism.
- Encryption: Every time the sender sends an encrypted message to the receiver she sends an OCSP query to the VA to ensure that the receiver's certificate is still valid.

Certificate Revocation Trees: Kocher suggested an improvement over OCSP [7]. Since the VA is a global service it must be sufficiently replicated in order to handle the load of all the validation queries. This means the VA's signing key must be replicated across many servers which is either insecure or expensive (VA servers typically use tamper-resistance to protect the VA's signing key). Kocher's idea is to have a single highly secure VA periodically post a signed CRL-like data structure to many insecure VA servers. Users then query these insecure VA servers. The data structure proposed by Kocher is a hash tree where the leaves are the currently revoked certificates sorted by serial number (lowest serial number is the left most leaf and the highest serial number is the right most leaf). The root of the hash tree is signed by the VA. This hash tree data structure is called a Certificate Revocation Tree (CRT).

When a user wishes to validate a certificate CERT she issues a query to the closest VA server. Any insecure VA can produce a convincing proof that CERT is (or is not) on the CRT. If n certificates are currently revoked, the length of the proof is $O(\log n)$. In contrast, the length of the validity proof in OCSP is $O(1)$.

Skip-lists and 2-3 trees: One problem with CRT's is that, every time a certificate is revoked, the entire CRT must be recomputed and distributed in its entirety to the various VA servers. A data structure allowing for dynamic updates would solve this problem since the secure VA would only need to send small updates to the data structure along with a signature on the new root of the structure. Both 2-3 trees proposed by Naor and Nissim [10] and skip-lists proposed by Goodrich [5] are natural data structures for this purpose. Additional data structures were proposed in [1]. When a total of n certificates are already revoked and k new certificates must be revoked during the current time period, the size of the update message to the VA servers is $O(k \log n)$ (as opposed to $O(n)$ with CRT's). The proof of certificate's validity is $O(\log n)$, same as with CRTs.

7.2 Comparison with SEM architecture

CRLs and OCSP are the most commonly deployed certificate revocation techniques. Some positive experiments with skip-lists are reported in [5]. We compare the SEM architecture with CRLs and OCSP. Since CRT's and skip-lists are used in the same way as OCSP (i.e., query a VA to obtain a proof of validity) most everything in our OCSP discussion applies to these methods as well.

Immediate revocation: Suppose we use CRLs for revocation. Then, Bob verifies a signature or encrypts a message he must first download a long CRL and verify that the Alice's certificate is not on the CRL. Note that Bob is uninterested in all but one certificate on the CRL. Nevertheless, he must download the entire CRL since, otherwise, the VA's signature on the CRL cannot be verified. Since CRLs and Δ-CRLs tend to get long, they are downloaded infrequently, e.g., once a week or month. As a result, certificate revocation might only take effect a month after the revocation occurs. The SEM architecture solves this problem altogether.

Suppose now that OCSP is usd for revocation. Whenever Bob sends email to Alice he first issues an OCSP query to verify validity of Alice's certificate. He then sends email encrypted with Alice's public key. The encrypted email could sit on Alice's email server for a few hours or days. If, during this time, Alice's key is revoked (e.g., because Alice is fired or looses her private key) there is nothing preventing the holder of Alice's private key from decrypting the email after revocation. The SEM solves this problem by disabling the private key immediately after revocation.

Implicit timestamping: Both OCSP and CRLs require the signer to contact a trusted time service at signature generation time to obtain a secure timestamp for the signature. Otherwise, a verifier cannot determine with certainty when the signature was issued. If binding semantics are sufficient, the time service is unnecessary when using the SEM architecture. Once a certificate is revoked, the corresponding private key can no longer be used to issue signatures. Therefore, a verifier holding a signature is explicitly assured that the signer's certificate was valid at the time the signature was generated.

Shifted validation burden: With current PKIs, the burden of validating certificates is placed on: (1) senders of encrypted messages and (2) verifiers of signed messages. In the SEM architecture, the burden of certificate validation is reversed: (1) receivers of encrypted messages and (2) signers (generators) of signed messages.

SEM **Replication (A disadvantage):** Since many users need to use the SEM for decryption and signing, it is natural to replicate it. However, replicating the SEM across organizations is not recommended for the same reason that replicating the VA in OCSP is not recommended. Essentially, the SEM generates tokens using a private key known only to the SEM. The result of exposing this key is that an attacker could unrevoke certificates. Replicating the SEM might make it easier to expose the SEM's key. Hence, the SEM architecture is mainly applicable in the same environments where OCSP is used, i.e., mainly medium-sized organizations. The SEM architecture is not geared towards the global Internet.

8 Conclusions

We described a new approach to certificate revocation. Rather than revoking the user's certificate our approach revokes the user's ability to perform cryptographic operations such as signature generation and decryption. This approach has several advantages over traditional certificate revocation techniques: (1) revocation is instantaneous – the instant the user's certificate is revoked the user can no longer decrypt or sign messages, (2) when using binding signature semantics there is no need to validate the signer's certificate during signature verification, and (3) using mRSA this revocation technique is transparent to the peer – the system generates standard RSA signatures and decrypts standards RSA encrypted messages.

We implemented the SEM architecture for experimentation purposes. Our measurements of the implementation show that signature and decryption times are essentially unchanged from the user's perspective. Therefore, we believe this architecture is appropriate for a medium-size organization where tight control of security capabilities is desired. The SEM architecture is not designed for the global Internet.

9 Acknowledgments

The authors gratefully acknowledge Carl Ellison for helping us to get this paper into proper shape and bringing up a number of interesting points. We also thank Paolo Montini and Ignacio Solis for their help with the initial SEM and mRSA prototyping.

References

[1] W. Aiello, S. Lodha, R. Ostrovsky, "Fast digital identity revocation", In proceedings of CRYPTO '98.

[2] D. Boneh, M Franklin, "Efficient generation of shared RSA keys", In Proceedings of Crypto' 97, Lecture Notes in Computer Science, Vol. 1233, Springer-Verlag, pp. 425–439, 1997.

[3] P. Gemmel, "An introduction to threshold cryptography", in CryptoBytes, a technical newsletter of RSA Laboratories, Vol. 2, No. 7, 1997.

[4] N. Gilboa, "Two Party RSA Key Generation", in Proceedings of Crypto '99.

[5] M. Goodrich, R. Tamassia, and A. Schwerin, "Implementation of an Authenticated Dictionary with Skip Lists and Commutative Hashing", In Proceedings of DARPA DISCEX II, June 2001.

[6] S. Haber, W.S. Stornetta, "How to timestamp a digital document", J. of Cryptology, Vol. 3, pp. 99–111, 1991.

[7] P. Kocher, "On Certificate Revocation and Validation", Financial Cryptography – FC '98, Lecture Notes in Computer Science, Springer-Verlag, Vol. 1465, 1998, pp. 172-177.

[8] M. Malkin, T. Wu, and D. Boneh, "Experimenting with Shared Generation of RSA keys", In proceedings of the Internet Society's 1999 Symposium on Network and Distributed System Security (SNDSS), pp. 43–56.

[9] S. Micali, "Enhanced certificate revocation system", Technical memo, MIT/LCS/TM-542b, March 1996.

[10] M. Naor, K. Nissim, "Certificate revocation and certificate update", In proceedings of USENIX Security '98.

[11] M. Myers, R. Ankney, A. Malpani, S. Galperin and C. Adams, "X.509 Internet PKI Online Certificate Status Protocol - OCSP". IETF RFC 2560, June 1999.

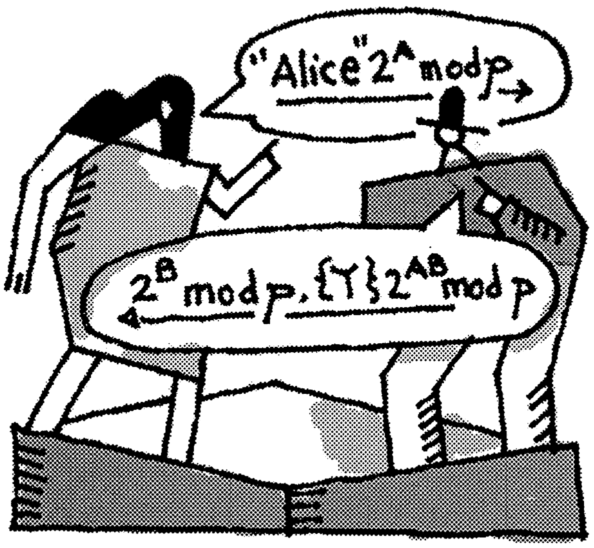
[12] OpenSSL, `http://www.openssl.org`

[13] R. Rivest, "Can we eliminate Certificate Revocation Lists", Financial Cryptography – FC '98, Lecture Notes in Computer Science, Springer-Verlag, Vol. 1465, 1998, pp. 178-183.

[14] R. Rivest, A. Shamir and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", CACM, Vol. 21, No. 2, February 1978.

[15] SEM Eudora plug-in. `http://crypto.stanford.edu/semmail/`

MATH ATTACKS!



Session Chair: Ian Goldberg, *Zero Knowledge Systems*

PDM: A New Strong Password-Based Protocol

Charlie Kaufman
Iris Associates
ckaufman@iris.com

Radia Perlman
Sun Microsystems Laboratories
radia.perlman@sun.com

Abstract

In this paper we present PDM (Password Derived Moduli), a new approach to strong password-based protocols usable either for mutual authentication or for downloading security information such as the user's private key. We describe how the properties desirable for strong password mutual authentication differ from the properties desirable for credentials download. In particular, a protocol used solely for credentials download can be simpler and less expensive than one used for mutual authentication since some properties (such as authentication of the server) are not necessary for credentials download. The features necessary for mutual authentication can be easily added to a credentials download protocol, but many of the protocols designed for mutual authentication are not as desirable for use in credentials download as protocols like PDM and basic EKE and SPEKE because they are unnecessarily expensive when used for that purpose. PDM's performance is vastly more expensive at the client than any of the protocols in the literature, but it is more efficient at the server. We claim that performance at the server, since a server must handle a large and potentially unpredictable number of clients, is more important than performance at the client, assuming that client performance is "good enough". We describe PDM for credentials download, and then show how to enhance it to have the properties desirable for mutual authentication. In particular, the enhancement we advocate for allowing PDM to avoid storing a password-equivalent at the server is less expensive than existing schemes, and our approach can be used as a more efficient (at the server) variant of augmented EKE and SPEKE than the currently published schemes. PDM is important because it is a very different approach to the problem than any in the literature, we believe it to be unencumbered by patents, and because it can be a lot less expensive at the server than existing schemes.

1 Introduction

This paper presents a new mechanism for allowing a user, armed only with a name and password, to connect to a network from a "generic" client machine, one loaded with software, but not with any user-specific configuration information (such as a private key or the public key of a trusted CA). The most secure solution to the problem of a human attaching via a generic workstation is a smart card. But until smart cards and readers become ubiquitous, and to handle the case when the user has left his smart card at home, there will still be a need for authentication based solely on something humans can remember and type, i.e., a password. Unfortunately, passwords are subject to dictionary attacks because most people are not willing to type and remember a sufficiently long and hard-to-guess password. So it is important to design a protocol in which even though passwords are used as keys, an eavesdropper or someone impersonating either the client or the server will not obtain information with which to do a dictionary attack.

There are several protocols in the literature for solving this problem. EKE [BM92] uses a Diffie-Hellman exchange encrypted with the user's password. SPEKE [Jab96] uses a function of the user's password as the base in a Diffie-Hellman exchange. Later, enhancements to both EKE and SPEKE were added to avoid storing a password-equivalent at the server [BM94], [Jab97]. SRP [Wu98], has the same properties as the augmented EKE and SPEKE, but better performance. AMP [Kwon01] is similar to SRP, with similar properties. [GL00] presents a protocol with similar properties using linear polynomials over GF(2^n), with a proof that the result is as secure as factorization. [BMP00] presents a 3-message variant of augmented EKE and proves it as secure as the Decision Diffie-Hellman (DDH) in the random oracle model. Similarly, [MS99] presents a protocol based on RSA, and proves the security of it based on the random oracle model. There is an unpublished protocol called S.N.A.K.E. by Peter Gunn that does Diffie-Hellman based not on a single strong prime, but a set of strong primes selected from a known set (of perhaps 2000), with the subset chosen being based on the password. [RCW98] presents a protocol (called S3P-RSA) in which an exponent is generated deterministically from a user's password, and used to transmit a strong secret. But this protocol has been shown to be broken, since many passwords can be tested simultaneously in an on-line attack.

[FK00] presents an additional interesting property that a strong password scheme might have, along with an algorithm for accomplishing that property. This property is the ability to break a user's strong secret into multiple pieces, such that theft of multiple servers' databases are required in order to do a dictionary attack. The disadvantage of this approach is that it requires multiple servers to be available or else the user will not be able to obtain his credentials, and it has lower performance because the user must do a protocol with multiple servers. Our protocol (PDM) does not have this property, and instead requires interaction with only a single server, which in many situations would be more desirable.

PDM (password derived moduli) does Diffie-Hellman based on a safe prime p ("safe" means a prime for which $(p-1)/2$ is also prime), where p is deterministically generated from the user's password, salted with information such as the user's name. A new approach, even if it gave no new functionality over old approaches, is still potentially important. Sometimes an approach will be found to have flaws, so alternatives are useful. Sometimes two approaches that seem to provide identical functionality are later found to have different properties in subtle ways. For example, although EKE and SPEKE seemed to provide identical functionality, [PK99] demonstrated that a 2-message protocol with salt was possible with SPEKE (with a composite modulus), and was not possible with EKE.

But PDM's performance properties make it a potentially important approach. PDM is vastly more computation intensive for the client than previous approaches, but since a client machine only needs to do the computation intensive operation once (per user, assuming the user has typed her password correctly), what is important is whether the performance is "good enough", which we claim it is. Although lower performance at the client is obviously a disadvantage, it has the beneficial side-effect of making on-line password guessing much slower.

Computation at the server is far more important, since a server might have to simultaneously deal with multiple clients. An attacker impersonating a client forces the server to do as much computation as legitimate clients. PDM can be vastly more efficient at the server since, with secret moduli, Diffie-Hellman must be broken *per password guess*. In most protocols, a single p is used for all users, so it is worth considerable effort to break Diffie-Hellman for that p. For most uses a Diffie-Hellman prime of less than 1000 bits would not be considered secure, but for PDM, a 500-bit prime might be sufficiently secure since breaking 500-bit Diffie-Hellman is estimated to require 8000 MIP-years - a high price to pay to test a single password guess for a single user. If PDM is sufficiently secure with a prime half as big, it will require 1/4 as much computation at the server as the best of any of the other schemes.

One natural worry is low-performance clients, such as hand-held devices. But those devices are carried by the user, and owned by the user, and therefore can be configured with a user-specific high-quality secret. Therefore such devices do not need schemes such as the one in this paper, which are naturally suited to the environment where there is an adequately powered workstation that has no configured information for the user.

In this paper we first describe how the desirable properties for a credentials download protocol differ from those for a mutual authentication protocol. Then we present the simplest PDM scheme, the one suitable for credentials download, or a mutual authentication protocol with the properties of EKE or SPEKE. As described in [Pat97] and [BM92], it is tricky to design such schemes so that an eavesdropper gains no information. We give an example of a potential vulnerability of EKE in section 3.2. In this paper we analyze our scheme for such vulnerabilities and design the protocol so that it does not leak information which would enable an eavesdropper to eliminate passwords.

Then we show how to enhance PDM to create a mutual authentication protocol that has higher performance at the server than existing password-based mutual authentication protocols (although it will still be more expensive at the client). This involves a method of avoiding storing a password-equivalent at the server. The scheme presented in this paper for accomplishing this is more efficient (for the server) than any previous scheme, even without the savings of using a smaller Diffie-Hellman modulus. Another enhancement is to prevent two servers from impersonating each other to a client that uses the same password on each of them. This enhancement works for any of the protocols, and has been proposed in the protocols in [BPR00], [MS99] and [GL00].

2 Properties of Credentials Download vs. Mutual Authentication Schemes

In general, desirable properties of a strong password scheme include:

- User Alice need only know her name and password.
- The workstation need not be configured with any user-specific security information (such as the public key of the server to which Alice will authenticate or download her credentials).
- An eavesdropper on an authentication exchange between user Alice and server Bob cannot learn Alice's password or be able to capture any information that could be used in an off-line password-guessing attack.
- Someone impersonating Alice to Bob, or Bob to Alice, will not be able to gain any information with which to do an off-line password-guessing attack, though one of them will be able to verify a single on-line guess.
- Bob's database should be *salted*, so that a dictionary attack against a stolen copy of the database would have to be launched separately per user, rather than computing hashes of all passwords in the dictionary, and comparing it against all users' information.

Additional properties desirable for a mutual authentication protocol, that are not necessary for a credentials download protocol are:

- Alice authenticates Bob. This is not necessary in credentials download because all Alice cares about is whether she's getting authentic credentials, not whether she's getting it from an authentic source.
- Bob authenticates Alice. This is not necessary in credentials download if the credential is encrypted with a high quality secret such that the requester cannot do a dictionary attack.
- An attacker will not be able to authenticate using replayed messages. This is not an issue in credentials download since all an attacker can do by replaying Alice's message is to get Bob to replay what he previously transmitted.
- Someone that steals Bob's database will not be able to directly use the information to impersonate Alice to Bob or any other server (there is no *password-equivalent* stored at Bob), though they will be able to use it to do an off-line password-guessing attack. (In credentials download, this is not important because if someone has stolen Bob's database he already has Alice's encrypted credential, which was also in the database, so being able to do Alice's piece of the protocol is not an advantage).
- If Alice uses the same password on multiple servers, say Bob and Ted, the information Bob stores cannot be used to impersonate Ted to Alice. (Again, in credentials download, it does not matter who gives the credential to Alice).

As a result of not needing these properties, a credentials download protocol can be simpler. It can be stateless for the server, have better performance, and require fewer messages (e.g., two). Once credentials are securely downloaded, the client can engage in any authentication protocol that assumes strong secrets and/or configured CA public keys (e.g. SSL, IPSec, Kerberos, SSH, etc.).

3 PDM: Password Derived Moduli

The key to our protocol, just as with EKE, SPEKE, and many descendents of these protocols, is to modify a Diffie-Hellman exchange with a function of the user's password. For example, EKE encrypts the Diffie-Hellman public number with a function of the user's password. SPEKE uses the user's password to calculate a base for the Diffie-Hellman exchange. We calculate a prime p that is a function of the user's password. This is done by using the user's password as a seed for a pseudo-random number generator that will be used in the search for an appropriate prime.

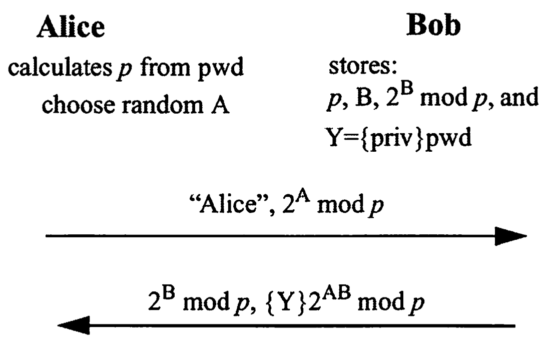
3.1 PDM for Credentials Download

We begin with just the simplest form of a strong password-based authentication protocol, that has only the functionality of the original EKE and SPEKE, and only the properties necessary for credentials download. On the surface, the protocol is extremely simple. The server Bob stores, for user Alice, p. We will always use 2 as the base. The reasons for using 2:

- it makes it easy to recognize small exponent cheating by someone impersonating the client (see section 3.2.2),
- to be different from SPEKE, to avoid potential patent infringement, and
- to use the law of quadratic reciprocity to choose candidate p's for which 2 is certain to be a generator (and thus avoid the performance cost of having to search for a generator, or the possible security loss as explained in section 3.4.2 of using a base that isn't a generator). If p is equal to 3 mod 8 and p is a *safe prime* ((p-1)/2 is also prime), 2 will be a generator.

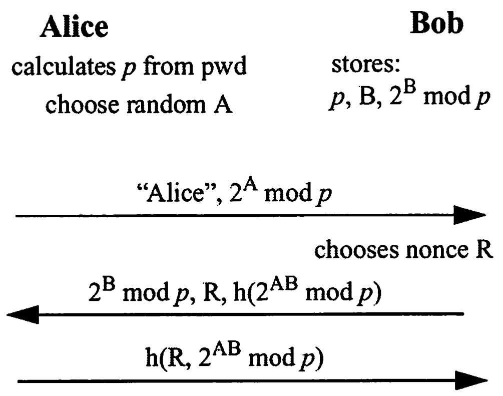
We chose a target of 10 seconds for a reasonable amount of computation time for the client to pick a p. Unfortunately, finding a safe p of a size considered secure for traditional Diffie-Hellman (say 1000 bits) would take longer (on today's typical client machines) than our target of 10 seconds for the user to log in. So as we discuss in section 3.4, there are various corners we can cut while maintaining good enough security for practical purposes. Indeed, on a 400 MHz processor, a 500-bit safe prime can be found within 10 seconds, and we argue that for our application, Diffie-Hellman with a 500-bit modulus would give adequate security because of the necessity for an eavesdropper doing a dictionary attack to break 500-bit Diffie-Hellman *for each guessed password.* And with the trick suggested in section 3.4.3, even a 1000-bit safe prime can be generated well within our budget of 10 seconds at the client. We give timing estimates for generating safe primes of various sizes in section 3.5.

For a simple 2-message credentials download protocol, the server Bob stores p and the credential Y, which is, for instance, the user's private RSA key encrypted with her password. The workstation calculates p from the user's password. As observed in [PK99], for credentials download it is possible to save Bob an exponentiation by having him always use the same B for Alice, and storing B and $2^B \bmod p$. (Note: the notation "{data}key" means "data" encrypted with key "key").



PDM for credentials download

For mutual authentication, especially if the rest of the session is not cryptographically protected by the resulting Diffie-Hellman key, then Bob can still save himself an exponentiation, but has to additionally furnish a nonce R in message 2, and Alice should return a function of both the nonce and $2^{AB} \bmod p$. Without the nonce an eavesdropper could replay Alice's messages and Bob would accept this as Alice having authenticated. So here is a mutual authentication PDM-based protocol in which Bob need only do one exponentiation. This scheme stores a password-equivalent at Bob, but in currently published schemes Bob requires more than 2 exponentiations.



Single exponentiation mutual authentication

This trick of saving Bob an exponentiation will not work if we want the additional feature of not storing a password-equivalent at Bob. Also, if the protocol is being used to establish a session key as well as just doing the initial authentication, perfect forward secrecy would be lost by having Bob always use the same B.

3.2 Avoiding Leaking Information

As discussed in both [BM92] and [Pat97] protocols such as these need to be implemented carefully or else information will be leaked. For instance, in the most straightforward implementation of EKE, one might encrypt $g^A \bmod p$ with a hash of the password. An eavesdropper that observed an encrypted $g^A \bmod p$ could do trial decryptions with various passwords, and eliminate any passwords in which the result was larger than p. If p was just a little more than a power of 2, then about half the passwords could be eliminated each time an eavesdropper observed a Diffie-Hellman value encrypted with a password. This might occur twice per authentication in variants of EKE that have both sides encrypting their Diffie-Hellman values.

3.2.1 Choosing p from a Small Range

In PDM, care must be taken to avoid allowing an eavesdropper to eliminate passwords based on seeing $2^A \bmod p$ and $2^B \bmod p$. If either transmitted Diffie-Hellman number was greater than the p derived from a candidate password, an eavesdropper could rule out that password.

We solve this problem by discarding A (or B) in the case where $2^A \bmod p$ is greater than the smallest possible p that could be derived from any password. To make the probability acceptably low that an A would have to be discarded (forcing an additional exponentiation) we choose p's from a very small range (e.g., if the smallest p and the largest p differ by less than 0.1%, then a Diffie-Hellman number will need to be rejected less than one time in 1000). We choose a p from a narrow range very close to a power of 2. We make it a narrow range by fixing the top 64 bits of the number at which our search will take place. Any constant will do, but to make maximal use out of the bits, the constant might as well be 63 1's followed by a 0. With a prime of, say, 700 bits, that gives a space of 700-64 bits, or 636 bits from which to choose p's, obviously large enough that there will be no shortage of p's, and yet the fraction of 700-bit space from which the p's are chosen is $1/2^{64}$. With this fraction, the probability of ever getting a $2^A \bmod p$ larger than the smallest possible p is $1/2^{64}$. And if it did occur, the only consequence is that authentication would take a little longer since another A would need to be chosen.

3.2.2 User Impersonator Picking Small A such that $2^A < p$

Another threat is that Trudy, impersonating Alice, could choose a very small A, such that 2^A would not be larger than p. Then Trudy could guess passwords based on what Bob sends, since Trudy has not committed to a value of p (because $2^A \bmod p$ has the same value for all possible p). In order to test a candidate password, Trudy needs to know the A corresponding to $2^A \bmod p$ for various values of p. If 2^A is less than p, then she knows such a pair for all p. If 2^A is even slightly more than p, and therefore needs to be reduced by p, she gets only a single pair.

Using the constant "2" for the base has the fortunate side-effect that it is very easy to detect if someone is cheating and sending a value that did not need to be reduced mod p. We require the sender to choose a Diffie-Hellman exponent larger than the log of p (i.e., if p is 700 bits long, then the exponent must be > 700) so that the result will need to be reduced by p. Since 2 is a generator, there cannot be two different exponents that yield the same value mod p. Therefore, if the number is of the form $10000...00000_2$, (i.e., the binary representation contains a single 1) then the sender has cheated by using an exponent sufficiently small that it did not need to be reduced by any modulus.

3.2.3 Timing Attacks

Because calculating p from a password involves searching for a prime at a pseudo-random value and testing until one is found, different passwords would take substantially different amounts of time to compute p. If an eavesdropper knew with some precision how long it took Alice's machine to compute p, this information could be used to eliminate many candidate passwords.

For example, a protocol which would give an eavesdropper timing information is one in which Alice's machine does not start computing p until it receives a message from Bob, perhaps because it needs to receive a salt value (see section 3.3) from Bob before it can compute p. The time until Alice's reply will be approximately the amount of time required for the machine to calculate p.

So it is best if Alice's workstation can compute p from the password before beginning the authentication protocol. This is possible if the salt is implicit, e.g. it is a canonical representation of the user name, since then the computation is done before messages are sent and an eavesdropper cannot time how long it took to compute p. A second choice, if implicit salt is not possible (too many variations on the name), would be to have Alice's typing of the password occur after she types the name of the server she wishes to contact. Since user typing times are highly variable, an eavesdropper will not be able to tell how much of the interval between Bob's message (e.g., sending salt), and Alice's machine's reply was due to computation of p and how much was due to Alice typing the password.

3.3 User Salt

It is highly desirable for user Alice's machine to be able to compute p before talking to the server, because:

- it will take the client machine a long time to compute p, so it would be good to be computing it while the user is doing other things, for instance, typing the name of the service she wishes to access.
- we don't want to allow an eavesdropper to tell how long it takes to compute p.
- If p is user/password dependent, but not server dependent, then a user can use the same p on multiple servers, ensuring that the expensive computation of p need only be done once per user, even if the user is using PDM for mutual authentication with multiple servers.
- it would take an extra message to send the salt.

In order to compute p before talking to the server, the salt value must be *intrinsic*, i.e., computable from information known locally about the user. Since this consists of the user's name and password, the logical choice for salt value is the user's name. It is important, however, to have a canonical version of the name. Capitalization or nicknames must not affect the computation of p.

3.4 Performance

The computation the server must perform to execute the basic PDM protocol (assuming equal sized moduli) is comparable to the best of the protocols with similar functionality even if the same size modulus is used. (This assumes that a protocol such as EKE or SPEKE is modified as suggested in [PK99] to have the server store B per user to save an exponentiation).

By using a different technique (as described in section 4) to achieve the goal of not storing a password equivalent at the server, PDM has better performance (even with the same sized modulus) than any of the previous schemes, though that technique could apply to EKE or SPEKE to make them equivalent in server performance (with the same size modulus). Although PDM is more expensive at the client than any of the prior protocols, we claim that since the client machine only needs to do the computation once, the only thing that matters in practice is for performance at the client to be "good enough". During the initial authentication, a human is waiting, and it is unacceptable for a user to wait for more than about 10 seconds to log in (and that's pushing it). Choosing p to be a 1000 bit safe prime would take more than a minute (see section 3.5) on today's typical desktop machine. Fortunately, there are some shortcuts we can take that raise performance dramatically. Note that as machines get faster we can drop more and more of the shortcuts.

3.4.1 Size of p

Today's conventional wisdom says that the size of a prime used in a Diffie-Hellman exchange should be on the order of 1000 bits. But given that this is not an ordinary Diffie-Hellman exchange, might a smaller prime be acceptable? Computation time falls dramatically with the size of the prime. What is the threat if our prime is smaller?

It is within the realm of possibility to break Diffie-Hellman with a size of, say, 500 bits, which at today's estimates would take on the order of 8,000 MIP-years. Eavesdropping on any authentication would yield a quantity with which password guesses could be verified, but it requires, for each guessed password, computing p and breaking Diffie-Hellman with that p. So an attacker would have to break 500-bit Diffie-Hellman *per password guess*.

Perfect forward secrecy would be endangered using Diffie-Hellman primes that are within the realm of possibility to crack, because if someone were to record conversations, and subsequently learn the user's password, then he'd be able to compute p, break 500-bit Diffie-Hellman, and then recover the session keys of the authentications that used that p. In practice, this is a sufficiently obscure threat that the size of the Diffie-Hellman prime is unlikely to be the weakest link in the chain (on-line password guessing, or using the learned password to directly impersonate the user in future conversations would probably be more fruitful), so in practice a 500 bit p might suffice. Alternately, and at some cost in complexity and server computation, the perfect forward secrecy attack could be circumvented by supplementing this protocol with a second anonymous Diffie-Hellman exchange with fixed adequate strength primes. If the result of that second Diffie-Hellman exchange contributes to the session key, perfect forward secrecy is preserved. And since computation with a small p is so efficient, the double Diffie-Hellman (with one large fixed p and one small, based-on-the-password p) would still be of comparable performance at the server to the best of existing schemes.

3.4.2 Non-Safe Prime

We can also save time in generating p by not requiring p to be a safe prime. The cost of breaking Diffie-Hellman is a function of both the size of p and the size of the largest prime factor of p-1. It is much faster to find a p with the property that $(p-1)/2$ isn't prime, but merely has a large prime factor. Although it is believed that Diffie-Hellman will be sufficiently secure with a p of this form, we run into a problem of finding a generator for p if p is not a safe prime, since without knowing the factorization of p-1 it is difficult (if not impossible) to determine whether a given g is a generator of the group. Traditional Diffie-Hellman does not need to assure that g is a generator. It only needs to assure that g generates a large subgroup of p. But for us, it is important that our g (which will be 2) is a generator. Otherwise, it might leak information to an eavesdropper. If the eavesdropper knew, for a particular password, that 2 was not a generator for the corresponding p, and then saw a value that 2 could not generate for that p, that password could be ruled out for that user.

3.4.3 User-Supplied Hint

Another method of increasing performance is to use a trick suggested by Jeff Schiller of giving the user a hint to tell the workstation, such as several of the bits of the selected p. If the user can't remember the hint, the workstation must test all candidate numbers. If the user misremembers the hint, then authentication will fail since the workstation will compute the wrong p. The user will recognize that it is probably the wrong hint since computing p will be as slow as without the hint.

How much will performance be improved? Assuming you've sieved for factors of p and $(p-1)/2$ up to 10,000, to get a safe prime of 512 bits you'd have to test, on average, 1600 numbers. On a 400 Mhz processor, a safe prime of 512 bits can be found within our budget of 10 seconds without the hint. Using the "hint" telling you, for instance, 6 bits of p, reduces computation by a factor of 64, making it under our target of 10 seconds even for 1024-bit safe primes. This hint could be in the form of a single character (using upper and lower case, numbers, and two more characters).

3.5 Measured Timing for Generating p

On a 400 MHz processor, using code that was not optimized for performance, the following table shows mean generation times with and without a six-bit user-supplied hint.

size of p	without hint	with hint
512	8.1 seconds	.11 seconds
768	34 seconds	.57 seconds
1024	111 seconds	1.8 seconds

Times for Generating p

Even more speedup could be attained with a larger hint, but of course this stretches the abilities of the human to remember the hint.

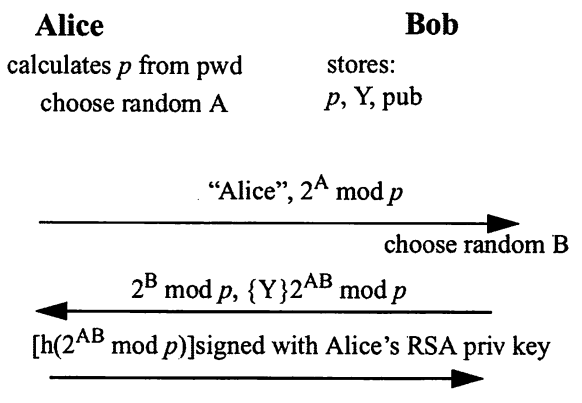
4 Avoiding a Password-Equivalent at the Server

In this section we discuss a different method of avoiding storing a password-equivalent at the server that is higher performance at the server than previous schemes. The approaches suggested in this section could be used for EKE and SPEKE as well, but not for AMP or SRP. The best previous method, SRP, involved doing two expensive exponentiations and one exponentiation with a 32-bit exponent. We present two new variants. The first is a little better in performance than SRP, assuming equal sized moduli, because our inexpensive exponentiation is an RSA verification (so the exponent could be as small as 3, rather than a 32-bit number as in SRP). The second involves only a single Diffie-Hellman exponentiation at the server and an RSA verification, so it is about half as much computation at the server as SRP, but it gives up perfect forward secrecy if someone steals Bob's database. (and again, this is assuming equal sized moduli).

In any of the schemes (ours as well as augmented EKE, SPEKE and SRP) it will be possible to do off-line password-guessing using a stolen copy of the server database, but without correctly guessing and verifying the password, the information in the server database would not be usable for impersonating the user to that (or any other) server.

The augmented versions of EKE and SPEKE, and protocols such as SRP and AMP do variants of having the server store $g^X \bmod p$ (where X is a function of the password), and require knowledge of X on the client. The augmented feature of these protocols requires an extra expensive exponentiation at the server.

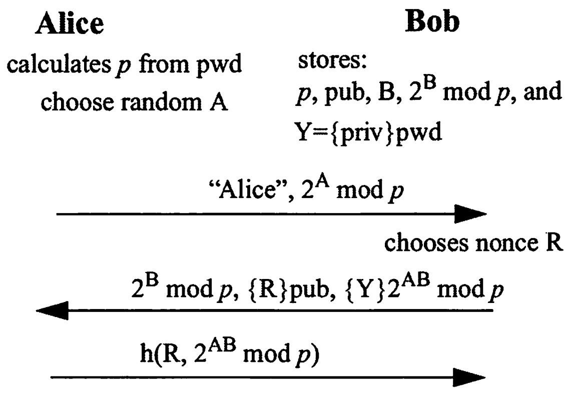
By using an RSA private key encrypted with Alice's password in place of $g^X \bmod p$ we can reduce the total computation for Bob to two expensive exponentiations and a single RSA public key verify, which can be very inexpensive (for example, if the public exponent is 3). Basing it on RSA is especially attractive because the same protocol works for download of an RSA private key as for mutual authentication. Bob stores p, Y (an RSA private key encrypted with the user's password), and pub (the associated public key). The protocol is:



No pwd-equivalent stored at server

In the protocol above, Bob has to compute two expensive exponentiations: raising 2 to B mod p, and raising (2^A mod p) to B mod p, and an inexpensive exponentiation (an RSA verify). This is slightly better in performance than the best previous scheme (SRP) because our inexpensive exponentiation, an RSA verify, is less expensive that SRP's inexpensive exponentation with a 32-bit exponent. It might also be the case with a secret modulus p (our scheme described in section 3) that the Diffie-Hellman exchange can be secure with a smaller p, which would further reduce the work for Bob. Note also that Alice must authenticate Bob. She does this by checking to see if Y, when decrypted, has the encoding of an RSA private key.

With the RSA-based scheme, we can reduce the work for Bob down to a single expensive exponentiation by allowing Bob to use the same B each time and adding a nonce as we did in section 3.1. If we make the session key be a function of the nonce as well as the Diffie-Hellman key, we can achieve "partial forward secrecy", a term we are using to mean someone would have to steal *both* Alice's private key and Bob's database in order to decrypt previous conversations.



Partial forward secrecy, single exponentiation

The session key should be some function of both the Diffie-Hellman key and R, such as h(1,R,2^{AB} mod p). We give up perfect forward secrecy because if someone steals *both* B and Alice's private key, they can decrypt a previously recorded conversation, since they will be able to compute 2^{AB} mod p (because they will have stolen B from Bob's database), and extract R (because of having stolen Alice's private key).

5 Preventing Servers from Impersonating Each Other to the User

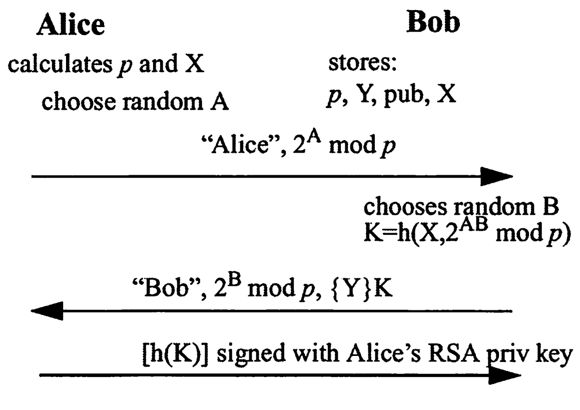
The third proposed enhancement is to prevent servers from impersonating each other to the user. If the information stored for user Alice is the same at server Bob as at server Carol, then Bob and Carol will be able to impersonate each other to Alice.

For this reason it is important to customize the information per server, so that even if Alice chooses the same password at multiple servers, the information at each will be different, and not usable to impersonate a different server to Alice.

The method of accomplishing this is to have some of the information stored for Alice be a function of the password and the server's name. It is desirable for Alice to have the same value for p at each server, since it is computation-intensive for Alice to compute p. So there should be some other quantity, X, that is a function of the server name. X will enable Bob to authenticate to Alice as "Bob" rather than as "any server on which user Alice has that password". Then even if the p is the same at Bob and Carol, they will not be able to impersonate one another to Alice because each only knows its own X.

So we suggest that p be computed using a seed which is solely a function of the user's name and user's password, and X be a function of the server's name, the user's name, and the user's password.

Bob stores p (generated from Alice's name and password, Y (Alice's private RSA key encrypted with her password), X (a hash of Alice's name, password, and Bob's name), and Alice's public key:



Prevent servers impersonating each other

6 Summary

In this paper we present PDM, a new method of doing strong password-based credentials download or mutual authentication. It has better performance at the server than any of the existing schemes, especially since it can use smaller moduli, because there is no single modulus on which the world could concentrate its Diffie-Hellman breaking efforts. Instead, Diffie-Hellman would have to be broken per user per password guess. We show that although performance at the client is far more expensive, that it is "good enough", especially with an optional user-supplied hint. We also present a method for avoiding a password equivalent which is less expensive than existing schemes at the server. This scheme could be applied to EKE or SPEKE, but not to schemes such as SRP and AMP that depend on everything being based on Diffie-Hellman. And we present a scheme with "partial forward secrecy" that is half as expensive as SRP, even with the same sized modulus.

Acknowledgments

We wish to thank Eric Rescorla for writing the code for generating PDM primes, and timing it for various sizes of p. We also wish to thank David Jablon for offering helpful comments.

References

[BM92] S. Bellovin and M. Merritt, "Encrypted Key Exchange: Password-based protocols secure against dictionary attacks", Proceedings of the IEEE Symposium on Research in Security and Privacy, May 1992.

[BM94] S. Bellovin and M. Merritt, "Augmented Encrypted Key Exchange: a Password-Based Protocol Secure Against Dictionary Attacks and Password File Compromise", ATT Labs Technical Report, 1994.

[BMP00] V. Boyko, P. MacKenzie, and S. Patel, "Provably Secure Password Authenticated Key Exchange Using Diffie-Hellman", Advances in Cryptology - EUROCRYPT 2000.

[BPR00] M. Bellare, D. Pointcheval, and P. Rogaway, "Authenticated Key Exchange Secure Against Dictionary Attacks", Advances in Cryptology - EUROCRYPT 2000.

[DH76] W. Diffie and M. Hellman, "New Directions in Cryptography", IEEE Transactions on Information Theory, November 1976.

[FK00] W. Ford and B. Kaliski, "Server-Assisted Generation of a Strong Secret from a Password", Proceedings of the IEEE 9th International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, 2000.

[GL00] O. Goldreich and Y. Lindell, "Session-Key Generation using Human Passwords Only", Cryptology ePrint Archive: Report 2000/057.

[Jab96] D. Jablon, "Strong password-only authenticated key exchange", ACM Computer Communications Review, October 1996.

[Jab97] D. Jablon, "Extended Password Protocols Immune to Dictionary Attack", Proceedings of the WETICE `97 Enterprise Security Workshop, June 1997.

[Kwon01] T. Kwon, "Authentication and Key Agreement via Memorable Password", ISOC NDSS Symposium, 2001.

[KPS95] C. Kaufman, R. Perlman, and M. Speciner, "Network Security: Private Communication in a Public World", Prentice Hall, 1995.

[MS99] P. MacKenzie and R. Swaminathen, "Secure Network Authentication with Password Identification", submission to IEEE P1363.

[Pat97] S. Patel, "Number Theoretic Attacks On Secure Password Schemes", Proceedings of the IEEE Symposium on Security and Privacy, May 1997.

[PK99] R. Perlman and C. Kaufman, "Secure Password-Based Protocol for Downloading a Private Key", ISOC NDSS Symposium, 1999.

[RCW98] M. Roe, B. Christianson, D. Wheeler, "Secure Sessions from Weak Secrets, Technical report from University of Cambridge and University of Hertfordshire, 1998.

[SS88] G. Steiner and J. Schiller, "Kerberos: An authentication service for open network systems", Proceedings of the USENIX Winter Conference, February 1988.

[Wu98] T. Wu, "The Secure Remote Password Protocol", ISOC NDSS Symposium, 1998.

Defending Against Statistical Steganalysis

Niels Provos*
Center for Information Technology Integration
University of Michigan
provos@citi.umich.edu

Abstract

The main purpose of steganography is to hide the occurrence of communication. While most methods in use today are invisible to an observer's senses, mathematical analysis may reveal statistical anomalies in the stego medium. These discrepancies expose the fact that hidden communication is happening.

This paper presents improved methods for information hiding. One method uses probabilistic embedding to minimize modifications to the cover medium. Another method employs error-correcting codes, which allow the embedding process to choose which bits to modify in a way that decreases the likelihood of being detected. In addition, we can hide multiple data sets in the same cover medium to provide plausible deniability.

To prevent detection by statistical tests, we preserve the statistical properties of the cover medium. After applying a correcting transform to an image, statistical steganalysis is no longer able to detect the presence of steganography. We present an *a priori* estimate to determine the amount of data that can be hidden in the image while still being able to maintain frequency count based statistics. This way, we can quickly choose an image in which a message of a given size can be hidden safely. To evaluate the effectiveness of our approach, we present statistical tests for the JPEG image format and explain how our new method defeats them.

*This research was supported in part by DARPA grant number F30602-99-1-0527.

1 Introduction

Steganography is the art and science of hiding the fact that communication is taking place. While classical steganographic systems depend on keeping the encoding system secret, modern steganography tries to be undetectable unless secret information is known, namely, a secret key. Because of their invasive nature, steganographic systems leave detectable traces within a medium's characteristics. This allows an eavesdropper to detect media that have been modified, revealing that secret communication is taking place. Although the secret content is not exposed, its hidden nature is revealed, which defeats the main purpose of steganography.

In general, the information hiding process starts by identifying redundant bits in a cover medium. Redundant bits are those bits that can be modified without destroying the integrity of the cover medium. The embedding process then selects a subset of the redundant bits to hold data from a secret message. The stego medium is created by replacing the selected redundant bits with message bits.

This paper presents two new methods to improve the selection process. The first method selects from a family of pseudo-random number generators [3]. Each pseudo-random number generator results in a different bit selection; the selection that causes the fewest changes to the cover medium is used for the embedding. The second method uses error-correcting codes. The result is greater flexibility in selecting bits. The two methods can be used together to minimize modifications to the cover medium.

Nonetheless, any modification of the redundant bits can change the statistical properties of the cover medium. For example, ones and zeros are equally likely in a message that has been encrypted. However, the redundant data being replaced might have a strong correlation towards either zero or one. Embedding the encrypted message weakens that correlation.

One way to prevent detection of steganographic content is to reduce the size of the hidden message. While such an approach decreases the likelihood of detection, it also results in decreased hiding capacity. Our paper presents a new method to preserve the statistical properties of a cover medium by applying additional transforms to the redundant data. The transforms correct measurable deviations in the statistics caused by the embedding process without decreasing the hiding capacity of the stego medium. We derive an *a priori* estimate for the amount of data that can be hidden while still being able to preserve frequency count based statistics. As a result, we can quickly identify images in which a particular message can be hidden safely.

While the method of using additional transforms is a generic concept that is data format independent, statistical properties and the specific transforms to preserve them depend on the data format of the stego medium. We illustrate existing statistical tests for the JPEG image format. Although these tests are not capable of detecting data embedded with our OutGuess [8] system, we present a new test that does detect the presence of steganographic content. We then demonstrate a specific transform for the JPEG format that preserves the image's statistical properties and thus prevents detection from statistical tests based on frequency counts.

The remainder of this paper is organized as follows. Section 2 introduces the prerequisites necessary for secure steganography and discusses related work in image steganography. In Section 3, we give an overview of the embedding process and introduce new methods to improve the embedding of hidden messages. After reviewing JPEG encoding in Section 4, we present statistical tests in Section 5. In Section 6, we show how to apply transforms that prevent detection by statistical tests. Section 7 provides an analysis of the transforms we use to correct deviations in the JPEG image format. We conclude in Section 8.

2 Prerequisites and Related Work

For steganography to remain undetected, the unmodified cover medium needs to be kept secret[1]. If it is exposed, a comparison between cover medium and stego medium immediately reveals changes. While an adversary gains knowledge of only approximately half of the embedded bits, she still detects modification.

Zöllner et al. [14] propose an information theoretic approach to solve the problem of secure steganography by employing nondeterministic selection. In their model, the original medium is known to the adversary but a preprocessing step introduces randomness into the cover medium. If the adversary can not obtain the transformed cover medium, she can not deduce information about the embedded message by observing differences between the original and the stego medium. In summary, they suggest two necessary conditions for secure steganography:

- The secret key used to embed the hidden message is unknown to the adversary.
- The adversary does not know the cover medium.

In practice, these two conditions are easily met. It suffices to create a cover medium with a digital camera or by scanning holiday pictures, and to discard the originals.

However, even though the original medium might not be available for comparison, the embedding process can introduce distortions. Analysis of many unmodified images may reveal characteristics that modified images lack. Identification of these characteristics allows us to perform correcting transforms after the embedding process that preserve the desirable characteristics.

Walton [12] authenticates an image by storing its checksum in the redundant bits of the image. The checksum is distributed uniformly over the image with a pseudo-random number generator. The probabilistic embedding in this paper differs by choosing a seed for the pseudo-random number generator that reduces the necessary changes to the cover medium.

[1]Throughout, we use the terminology established by Pfitzmann et al. [7].

Aura [2] uses a pseudo-random permutation generator to select the bits in the cover-medium. He notes that if the secret key and cover size remain unchanged, then the selected bits will be the same. In our embedding process, however, the pseudo-random number generator is reseeded to find the best embedding; in addition, the bit selection depends on the hidden message size.

Johnson and Jajodia [4] analyze images created with available steganographic software. Although they claim that current steganographic techniques leave noticeable distortions in the discrete cosine transform (DCT) coefficients, they do not further discuss the nature of these distortions.

Westfeld and Pfitzmann [13] describe visual and statistical attacks against common steganographic tools. They discuss ways that common steganographic techniques change statistical properties in the cover medium. For example, they evaluate one particular program that embeds data in JPEG images. To detect hidden information embedded by the program, they use a χ^2-test [6], which estimates the color distribution of an image carrying hidden information and compares it against the observed distribution.

The χ^2-test is perhaps too discriminating, in that it detects only programs that embed hidden message bits without spreading them over all redundant bits. In particular, this test does not detect the OutGuess embedding process, presented in Section 3.

In Section 5, we describe an extended χ^2-test that is capable of detecting more subtle changes. Even so, the methods developed in this paper prevent detection by both the original and the extended χ^2-test.

3 Embedding Process

The specific transforms we introduce to perform statistical corrections depend on embedding methods that distribute a hidden message over all redundant bits. This section explains the underlying steganographic embedding process and introduces new methods to improve on it.

We divide the task of embedding hidden information in a cover medium into two steps:

- Identification of redundant bits. Redundant bits can be modified without detectably degrading the cover medium.
- The selection of bits in which the hidden information should be placed.

Separating the embedding process into two parts allows for easy replacement. A different data format can be accommodated with a different identification algorithm, and new selection strategies can be implemented without changing other parts of the system. In addition, the computational cost of the selection does not depend on the cost of identifying the redundant bits.

One valid objection against this separation is potential loss of information that might be helpful in the selection step. For example, an image may have areas of high complexity that can either hold more hidden information or in which modifications are less likely to be detected. In our model, the selection algorithm sees only the redundant bits and is not aware of their origin. To remedy this, the identification step adds attributes to each redundant bit. These attributes indicate if a bit is locked or how detectable changes to it are.

3.1 Identification of Redundant Bits

In general, identifying the redundant bits of a data source depends on the specific output format. One has to be aware that the embedding actually happens when the cover medium is written out in that format. Conversion to the final data format might include operations like compression, and is not necessarily deterministic. Minimizing modifications to the cover medium requires knowledge of the redundant bits before the actual stego medium is created. For example, the OutGuess [8] system performs all operations involved in creating the output object and saves the redundant bits encountered. For the JPEG image format, this might be the LSB of the discrete cosine transform coefficients; see Section 4.

The hidden information overwrites the redundant bits when the final output is created. This requires determinism in the conversion process, which can always be ensured by replacing random processes with a pseudo-random number generator that is initialized to the same state for the identification and the final conversion step.

Before the identified bits are passed to the selection step, they are annotated with additional information. This information includes locked bits, *i.e.* bits that may not be modified in the embedding process, and a heuristic that determines how detectable changes to a bit might be.

A bit is locked when the bit has already been used to carry hidden information. This can occur when more than one message is hidden in the cover medium.

3.2 Selection of Bits

Before the selection of redundant bits can begin, an RC4 stream cipher [9] is initialized with a user-chosen secret key. We use the keyed stream cipher to encrypt the hidden message and derive a pseudo-random number generator (PRNG) for the selection process from it. The bits that are replaced with information from the hidden message are selected with the help of the pseudo-random number generator as follows.

First, we need to hide 32 state bits. The state is a concatenation of a 16-bit seed and a 16-bit integer containing the length of the hidden message. By varying the seed the selection can find a better embedding. Selection starts at the beginning of the identified bits. We determine the next bit by computing a random offset within a fixed interval and adding that offset to the current bit position. To compute the random offsets, we use the pseudo-random number generator described earlier. Data at the new bit position is replaced with the message data. This process is iterated 32 times. The resulting bit positions can be represented as,

$$\begin{aligned} b_0 &= 0 \\ b_i &= b_{i-1} + R_i(x) \quad \text{for } i = 1, \ldots, n \end{aligned}$$

where b_i is the position of the i-th selected bit, and $R_i(x)$ is a random offset in the interval $[1, x]$.

After the state data has been embedded, the pseudo-random number generator is reseeded with the 16-bit seed. The remaining length of the hidden message is used to adapt the interval out of which the random numbers are drawn to the amount of remaining data,

$$\text{interval} \approx \frac{2 \times \text{remaining redundant bits}}{\text{remaining length of message}}.$$

The selection process continues as outlined above, the only difference being that the interval is adjusted every eight bits. This way the hidden message is distributed evenly over all available bits.

Choosing the interval in this way restricts the hidden message size to a maximum of 50% of the available redundant bits. We explain in Section 6 why this is not a serious restriction. Not using all the redundant bits gives the selection process a greater opportunity to find a good embedding, as described in Section 3.3. It also leaves enough bits for the correcting transform to preserve frequency count based statistics.

Because the PRNG is keyed with a secret, it is not possible to find the hidden message without knowing the key. The recipient initializes the PRNG with that secret and uses the same selection process to retrieve the hidden message from the stego medium. The interval size is changed only after the state has been embedded, so the state is retrievable and can be used to reseed the pseudo-random number generator correctly.

3.3 Beneficial Reseeding of the PRNG

We now explain how modifications to the cover medium can be reduced with the selection algorithm from the previous section.

In the selection process, the keyed pseudo-random number generator can be reseeded with a freely chosen 16-bit seed. In effect, the seed creates a family of independent pseudo-random number generators. Each pseudo-random number generator selects its own subset of redundant bits. The selections result in a different number of bits that have to be modified. The distribution of the number of changed bits is binomial. The probability that k of n bits will be changed is as follows,

$$p_k^{(n)} = \binom{n}{k} p^k (1-p)^{n-k},$$

where p is the probability that a selected bit in the redundant data has to be changed.

As the hidden data has been encrypted by the RC4 stream cipher, it has the properties of a random stream, so we expect that $p = \frac{1}{2}$, and that the average number of changed bits will be close to 50% of the bits in the hidden data.

Picking a seed that represents the changed bits at the lower end of the binomial distribution allows us to reduce the number of bits that have to be modified; see Figure 1. It becomes harder to detect the modifications, as more of the hidden message is already naturally represented in the redundant bits.

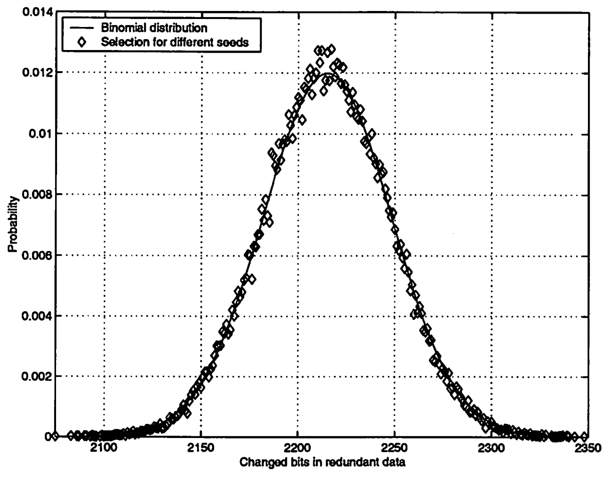


Figure 1: Probability distribution of changed bits for different seeds compared to a binomial distribution with $n = 4430$ and $p = \frac{1}{2}$.

Detectability is also used as a bias in the selection process. The selector does not try to reduce only the number of changed bits but also the overall detectability. Whenever a bit has to be modified, its detectability will be added to a global bias. A higher accumulated bias reduces the likelihood that this specific embedding will be used.

The standard deviation of a binomial distribution is given by $\sigma = \sqrt{npq}$ with $q = 1 - p$; in our case $\sigma = \frac{1}{2}\sqrt{n}$. We notice that the ratio of the standard deviation and the square root of the number of bits in the hidden message remains constant,

$$\frac{\sigma}{\sqrt{n}} = p.$$

As a result, more modifications can be avoided by reseeding the PRNG for smaller hidden messages than for bigger ones.

Bits	Standard Deviation σ		Probability
n	Expected	Measured	$\sigma/\sqrt{n}$
1216	17.436	16.193	0.464
2400	24.495	24.254	0.495
4176	32.311	32.205	0.498
6840	41.352	40.163	0.486
9800	49.497	44.257	0.447
14832	60.893	53.123	0.436

Figure 2: Deriving the probability p that a selected bit has to be changed from the size of a hidden message in bits n and the standard deviation σ for embedding into a JPEG-image with 77135 redundant bits.

Measuring the standard deviation of changed bits encountered in the selection step allows us to verify the probability p that a selected bit has to be modified as shown in Figure 2.

Furthermore, by varying the seed the algorithm is able to find an embedding that does not have conflicts with locked bits. This allows us to hide multiple data sets within a cover medium, which we discuss in Section 3.5.

3.4 Choices with Coding Theory

While the seeding allows us to view the embedding as a probabilistic process, the flexibility of the PRNG family in selecting bits is not always enough to avoid all bits that are locked or have a high detectability. We could prevent modification of those bits if it were possible to introduce errors into our hidden message without destroying its content. In other words, all introduced errors need to be correctable.

Coding theory provides us with codes that can correct errors by maximum-likelihood decoding. We write $[n, k, d]$ to indicate a k-dimensional linear code of length n with Hamming distance d. Such a code can correct t errors, where $d = 2t + 1$.

In general, the application of an error-correcting code increases the data that we need to embed. A k-bit data block will result in an n-bit code block. However, we first observe that about one half of the data is already represented and does not need to be modified, and second, that we can introduce t additional errors in the code block. Thus, if

$$\frac{n}{2} - t = \frac{k}{2} \qquad (1)$$

holds, we expect the number of modifications for the encoded data to be the same as for the unencoded data. Equation (1) can be rewritten as

$$d = n - k + 1,$$

which is exactly the Singleton bound fulfilled by all maximum distance separable (MDS) codes [10].

Unfortunately, the only non-trivial binary MDS code is the repetition code $[n, 1, n]$. It's major drawback is the n-fold repetition of the data, so that it is useful only for small hidden messages.

Once the data is encoded, we can choose t bits in each code block that do not need to be modified in the redundant data. Our choice is determined by conflicts with already locked bits from a previously embedded message, and after all conflicts have been resolved by the detectability of changes.

The approach that we take here is similar to the parity encoding suggested by Anderson [1]. However, using error-correcting codes has advantages over using the parity encoding. We can trade security against capacity by choosing a code that is not MDS. Additionally with error-correcting codes only $n - t$ bits are locked, whereas the parity encoding locks all n bits.

3.5 Plausible Deniability

To embed a hidden message into the cover medium, we modify the redundant data of the cover medium. The redundant data might have properties of a statistical nature of which we are not aware, or understand less well than an adversary. If the embedding process changes the characteristics of the cover medium, a more knowledgeable observer can conclude the presence of a hidden message without necessarily being able to point to the specific bits that were changed.

The originator of the stego medium might now be forced to reveal the hidden communication. However, we assume that the only predicate the observer can ascertain is the fact that the cover medium was modified. If the sender embeds multiple hidden messages into the cover medium, he can include an innocuous message, turn that over on request, claim that there is no other information hidden in the stego medium, and leave unharmed. This is called *plausible deniability.*

Actually, the described mechanism already implicitly supports plausible deniability. More than one hidden message can be embedded, as the "locked bit" attribute prevents information from other hidden messages from being overwritten. Depending on the size of the hidden messages, the likelihood that a selection is found that does not conflict with previously locked bits can be small. In that case, error-correcting codes can be employed to increase the selection flexibility.

3.6 Hidden Message Determines Cover

Usually, a hidden message is embedded into a specific medium. Instead of selecting a specific cover medium, the hidden message can be examined and a cover medium will be selected that allows embedding with minimal modifications.

This can be achieved by embedding the hidden message into multiple cover media. Afterwards, the cover medium that results in the fewest modifications is chosen.

As expected, the distribution of changed bits in the different cover media follows a binomial distribution similar to the one shown in Section 3.3.

4 JPEG image format

While the embedding methods mentioned in this paper are independent of the actual data format of the cover medium, each data format has its own statistical properties. We restrict our analysis to a very data format: JPEG [11]. However, similar characteristics can also be found in other formats. The general idea of correcting statistical deviations still applies, but requires different, appropriate transforms.

The JPEG image format uses a discrete cosine transform (DCT) to transform successive 8×8-pixel blocks of the image into 64 DCT coefficients each. The DCT coefficients $F(u, v)$, of an 8×8 block of image pixels $f(x, y)$, are given by

$$F(u,v) = \frac{1}{4}C(u)C(v)\Big[\sum_{x=0}^{7}\sum_{y=0}^{7} f(x,y) * \cos\frac{(2x+1)u\pi}{16}\cos\frac{(2y+1)v\pi}{16}\Big],$$

where $C(u), C(v) = 1/\sqrt{2}$ when u and v equal 0 and $C(u), C(v) = 1$ otherwise.

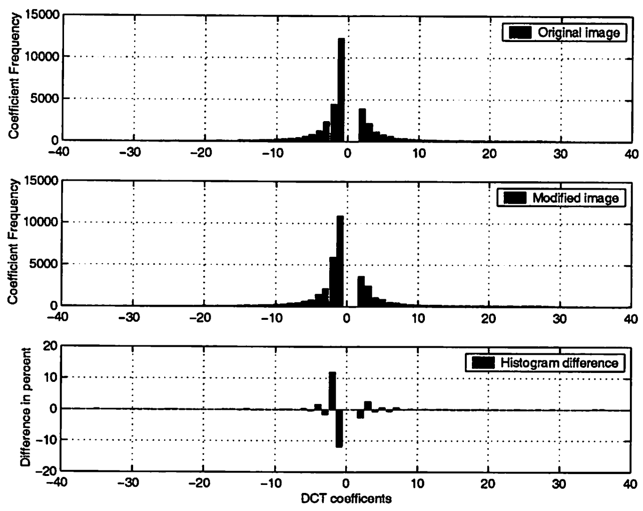


Figure 3: Differences in the DCT histograms are noticeable when the embedding process does not make any statistical corrections.

Afterwards the coefficients are quantized by the following operation:

$$F^Q(u,v) = \text{IntegerRound}\Big(\frac{F(u,v)}{Q(u,v)}\Big),$$

where $Q(u,v)$ is a 64-element quantization table.

The least-significant bits of those quantized DCT coefficients, for which $F^Q(u,v) \neq 0$ and $\neq 1$, are used as redundant bits into which the hidden message is embedded.

5 Statistical Tests

Statistical tests can reveal if an image has been modified by steganography. These tests determine if an image's statistical properties deviate from the norm. Some tests are independent of the data format and just measure the entropy of the redundant data.

The simplest test is to measure the correlation towards one. A more sophisticated one is Ueli Maurer's "Universal Statistical Test for Random Bit Generators" [5]. We expect images with hidden data to have a higher entropy than those without.

Westfeld and Pfitzmann outline an interesting statistical attack [13]. They observe that for a given image, the embedding of encrypted data changes the histogram of color frequencies in a particular way.

In the following, we clarify their approach and show how it applies to the JPEG format. In their case, the embedding process changes the least significant bits of the colors in an image. The colors are addressed by their indices in the color table. If n_i and n_i^* are the frequencies of the color indices before and after the embedding respectively, then the following relation is likely to hold

$$|n_{2i} - n_{2i+1}| \geq |n_{2i}^* - n_{2i+1}^*|.$$

In other words, the frequency difference between adjacent colors is reduced by the embedding process. In an encrypted message, zeros and ones are equally distributed. For $n_{2i} > n_{2i+1}$ that means that the bits of the hidden message change n_{2i} to n_{2i+1} more frequently than the other way around.

The same is true in the case of the JPEG data format. Instead of measuring the color frequencies, we observe differences in the frequency of the DCT coefficients. Figure 3 displays the histogram before and after a hidden message has been embedded in a JPEG image. The histogram differences are displayed in the subgraph at the bottom of the figure. We observe a reduction in the frequency difference between coefficient -1 and its adjacent DCT coefficient -2. Adjacent means that the coefficients differ only in the least significant bit. A similar reduction in frequency difference can be observed between coefficients 2 and 3.

Westfeld and Pfitzmann use a χ^2-test to determine whether the color frequency distribution in an image matches a distribution that shows distortion from embedding hidden data. In the following, we outline their test for the DCT coefficients in a JPEG. Because the test uses only the stego medium, the expected distribution y_i^* for the χ^2-test has to be computed from the image. The assumption for a modified image is that adjacent DCT frequencies are similar. Let n_i be the DCT histogram, we then take the arithmetic mean,

$$y_i^* = \frac{n_{2i} + n_{2i+1}}{2},$$

to determine the expected distribution and compare against the observed distribution

$$y_i = n_{2i}.$$

The χ^2 value for the difference between the distri-

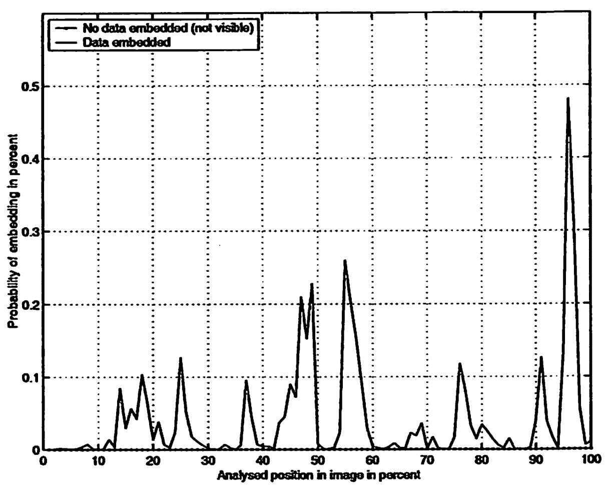


Figure 4: An extended χ^2-test where each sample covers 3.2% of the DCT coefficients detects the embedding in the modified image, but does not react to an unmodified image.

butions is given as

$$\chi^2 = \sum_{i=1}^{\nu+1} \frac{(y_i - y_i^*)^2}{y_i^*},$$

where ν are the degrees of freedom, that is, the number of different categories in the histogram minus one. It may be necessary to sum adjacent values from the expected distribution and from the observed distribution to ensure that there are enough counts in each category. Westfeld and Pfitzmann require that each count is greater than four. If two adjacent categories are summed together, the degrees of freedoms need to be reduced by one.

The probability p that the two distributions are equal is given by the complement of the cumulative distribution function,

$$p = 1 - \int_0^{\chi^2} \frac{t^{(\nu-2)/2} e^{-t/2}}{2^{\nu/2}\Gamma(\nu/2)} dt,$$

where Γ is the Euler Gamma function.

The probability of embedding is determined by calculating p for a sample from the DCT coefficients. The samples start at the beginning of the image and for each measurement the sample size is increased.

Because the test uses an increasing sample size and always starts at the beginning of the image, it detects changes only if the frequency histogram is distorted continuously from the beginning of the image. Intermediate areas in the image that do not exhibit distortions can cause negative test results. This is the case even if other areas in the image are clearly distorted. For this reason, the test does not detect the embedding process described in this paper.

5.1 Detection

It is possible to extend Westfeld and Pfitzmann's χ^2-test to be more sensitive to partial distortions in an image. Observe that two identical distributions produce about the same χ^2 values in any part of the distribution. Instead of increasing the sample size and applying the test at a constant position, we use a constant sample size but slide the position where the samples are taken over the entire range of the image. Using the extended test we are able to detect our simple embedding process; see Figure 4.

In this case, the sample size is set to 3.2% of all DCT coefficients. The test starts at the beginning of the image, and the position is incremented by one percent for every χ^2 application. This extended test does not react to an unmodified image, but detects the embedding in some areas of the stego image.

To find an appropriate sample size, we choose an expected distribution for the extended χ^2-test that should cause a negative test result. Instead of calculating the arithmetic mean of coefficients and their adjacent ones, we take the arithmetic mean of two unrelated coefficients,

$$y_i^* = \frac{n_{2i-1} + n_{2i}}{2}.$$

A binary search on the sample size is used to find a value for which the extended χ^2-test does not show a correlation to the expected distribution derived from unrelated coefficients.

6 Correcting Statistical Deviations

Not all of the redundant bits are used when embedding the hidden message. In fact, the selection process allows no more than half of the redundant bits to be used for data.

If we know what kind of statistical tests are being used to examine an image for modification, we

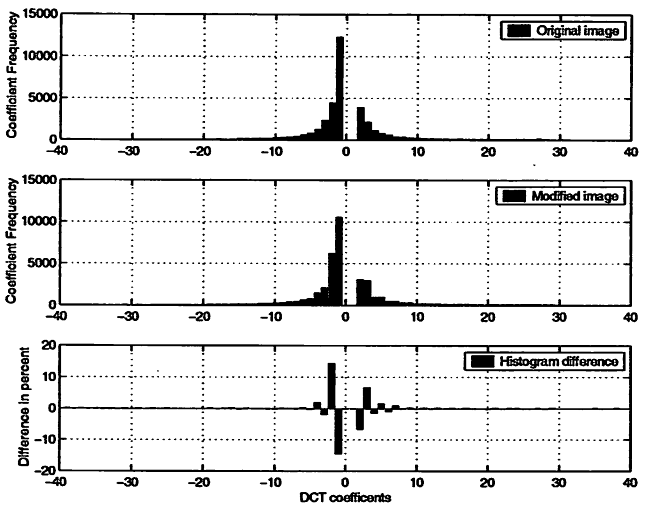


Figure 5: The naive statistical corrections cause the frequency of adjacent DCT coefficients to be equalized. It is immediately evident that the image is modified.

can use the remaining redundant bits to correct any statistical deviation that the embedding process created.

Our first (naive) approach included preserving the correlation to one and the entropy measured by the Maurer test. Essentially, when a bit is changed from zero to a one, we try to change a nearby bit from one to zero. Although, this approach helps to prevent entropy increase in the redundant data, it completely neglects statistics that depend on macroscopic properties. For the JPEG format, the result is a distortion in the DCT histogram, illustrated in Figure 5. The DCT coefficients -2 and -1 are even closer together than in Figure 3, and the frequencies for DCT coefficients 2 and 3 are nearly the same.

If we want to avoid distortions in the DCT histogram, additional corrections are necessary to maintain the distribution of the DCT coefficients. For example, suppose embedding a hidden message modifies the j-th DCT coefficient, $DCT(j)$. If $DCT(j) = 2i$, it will be modified to $2i + 1$. We correct this change by finding an adjacent coefficient $DCT(k)$, that is $DCT(k) = 2i + 1$, and changing it to $2i$. If we correct every change to the DCT coefficients, their histogram will be identical to the one of the original image.

Furthermore, a correcting transform that essentially swaps values keeps all frequency counts constant. Thus, no statistic that is based purely on frequency counts will be able to detect a difference between the original and the stego medium.

We make the following observation for frequency count based statistics. Let f be a frequency count in the histogram, and $\bar{f}$ its adjacent count. Without loss of generality, let $f \geq \bar{f}$. Let α denote the fraction of redundant bits that are used to hold the hidden message. After embedding, we expect the following changes in frequencies:

$$\begin{aligned} f^* &= f - \frac{\alpha}{2}(f - \bar{f}), \\ \bar{f}^* &= \bar{f} + \frac{\alpha}{2}(f - \bar{f}). \end{aligned}$$

In order for the transform to be able to correct the frequency count, enough unmodified coefficients need to be left in $\bar{f}$ so that the change in f can be adjusted, in other words the relation

$$(1 - \alpha)\bar{f} \geq \frac{\alpha}{2}(f - \bar{f})$$

must hold.

The relation yields an *a priori* estimate for the fraction α of redundant bits that can be used for data while still having enough bits left for the correcting transform to work:

$$\alpha \leq \frac{2\bar{f}}{f + \bar{f}}.$$

Given a hidden message, we can use the estimate to choose an image for which the correcting transform will be able to preserve the original frequency counts. Interestingly enough, for JPEG the fraction of redundant bits that can be used to hold the hidden message does not increase linearly for images with more DCT coefficients; see Figure 6.

The correcting transform has the following requirements:

1. For any part of the image, the distribution of the DCT coefficients should be similar to the unmodified image.
2. The number of corrections necessary to preserve statistical properties should be small.

Some statistical properties of the DCT coefficients may be unknown to us, so we try to prevent introducing additional distortions. Such distortions can

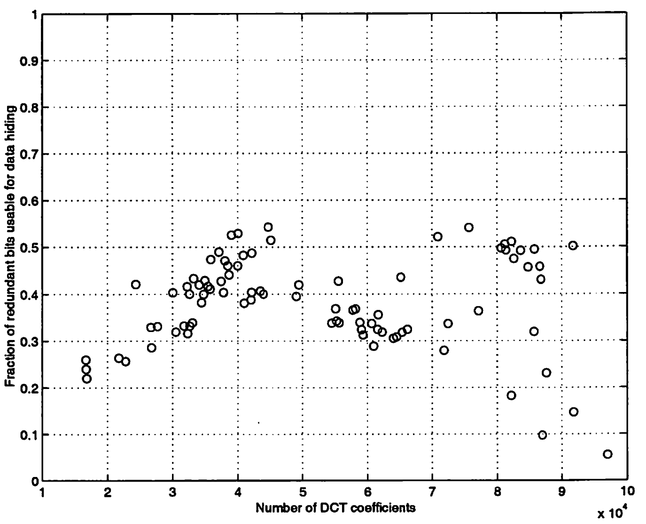


Figure 6: The fraction of the DCT coefficients that can be used for data hiding does not increase linearly for images with more coefficients.

result from corrections meant to preserve the statistics that we do know about. If we keep the number of additional modifications small, we reduce the likelihood of further distorting the image's statistical properties.

Furthermore, if steganography is to remain undetected by the extended χ^2-test, all parts of the image must be free of statistical distortions. The test will detect no embedding if each part of the modified image has a DCT coefficient distribution similar to the original.

Algorithm 1 meets both requirements. It is run after the embedding process finishes. In step 1, we compute the DCT frequency histogram from the original image and store it in N. Step 2 determines the threshold frequencies. The threshold indicates how many errors in the histogram we are willing to tolerate for a specific DCT coefficient. It is calculated by multiplying the observed frequencies of the DCT coefficients with the scaling factor α. When the number of errors for a coefficient exceeds its threshold, we modify the image to preserve the statistics for that coefficient.

Step 3 finds *AdjDCT*, the index of the coefficient adjacent to the modified one. In step 4, we determine if there are pending errors for the adjacent coefficient that should be corrected. In that case, the correction for the current DCT coefficient can be traded against the pending correction of its adjacent coefficient.

```
1 $N \leftarrow DCTFreqTable(original)$;
  $k \leftarrow$ number of coefficients in image;
2 $\alpha \leftarrow 0.03 * 5000/k$;
  for $i \leftarrow DCT_{min}$ to $DCT_{max}$ do
      $N_{\text{error}}[i] \leftarrow 0$;
      $N^*[i] \leftarrow \alpha N[i]$;
  endfor
  for $i \leftarrow 1$ to $k$ do
      if $DCT(i)$ unmodified then
          continue in loop;
      endif
3     $AdjDCT \leftarrow DCT(i) \oplus 1$;
4     if $N_{error}[AdjDCT]$ then
          decrement $N_{\text{error}}[AdjDCT]$;
          continue in loop;
      endif
5     if $N_{error}[DCT(i)] < N^*[DCT(i)]$ then
          increment $N_{\text{error}}[DCT(i)]$;
          continue in loop;
      endif
      if $exchDCT(i, DCT(i))$ fails then
          increment $N_{\text{error}}[DCT(i)]$;
          continue in loop;
      endif
  endfor
  for $i \leftarrow DCT_{min}$ to $DCT_{max}$ do
      while $N_{error}[i] \neq 0$ do
          decrement $N_{\text{error}}[i]$;
          $exchDCT(k, i)$;
      endw
  endfor
```

Algorithm 1: This transform preserves the statistical properties of an JPEG image. It keeps track of differences in the frequency counts between original and stego medium. If the differences exceed a certain threshold, the frequency count is adjusted.

If that is not the case, we check in step 5 if the number of errors for the coefficient, $N_{\text{error}}[DCT(i)]$, can be incremented without exceeding its threshold value. If another increment is possible, we continue with the next modification. Otherwise, we have to correct the current modification in the image. The *exchDCT* algorithm is responsible for that. If that fails too, we just go ahead and increase the error for the coefficient above the threshold and take care of it later.

After all modifications have been examined, we need to correct all remaining errors. Not all the corrections might be possible. However, if we are able to correct most of the errors, changes in the histogram are not detectable.

The *exchDCT*() algorithm is very simple. Given a coefficient value *DCT* and a position *i* in the image, it tries to find the same coefficient at a prior position and change it to its adjacent coefficient. It starts searching near the coefficient that caused the algorithm to be executed and works its way to the beginning of the image. Coefficients that hold data from the hidden message or that have been used for previous corrections are skipped by *exchDCT*(). The algorithm indicates success or failure.

```
Function: exchDCT()
Data    : $i, DCT$
$AdjDCT \leftarrow DCT \oplus 1$;
for $j \leftarrow i - 1$ to 1 do
    if $DCT(j) = DCT$ and
    $DCT(j)$ does not hold data and
    $DCT(j)$ has not been used for corrections
    then
        $DCT(j) \leftarrow AdjDCT$;
        return success
    endif
endfor
return failure
```

Algorithm 2: Find a specific DCT coefficient and change it to its adjacent DCT coefficient.

7 Analysis

To evaluate our correction algorithm, we embedded data into 54 pictures taken with a Fuji MX-1700 digital camera around Ann Arbor, Michigan. The size of the images is 640 × 480 pixels. After the images were downloaded from the camera, they were recompressed with a quality factor of 75. This simulates the conversion step in the embedding process without actually embedding any data.

For this set of images, the average number of DCT coefficients that we can use for modification is about 46,000, varying between 30,000 and 97,000. Each of these contributes one redundant bit.

Without embedding any data in the redundant bits, we notice a strong correlation towards one. On average, 63.8% of all the bits are set with a standard deviation of ±3.4% between images.

Method	One-Correlation	Maurer Test
All images		
Unmodified	63.41% ± 3.50%	6.732 ± 0.233
No corrections	59.10% ± 3.19%	6.976 ± 0.168
Corrections	62.91% ± 3.36%	6.775 ± 0.231
Images for which *a priori* estimate holds		
Unmodified	63.06% ± 3.53%	6.738 ± 0.241
Corrections	63.06% ± 3.53%	6.752 ± 0.231

Figure 7: Comparison between unmodified images, images with data embedded but without statistical corrections, and finally images with data embedded plus statistical corrections.

We embedded the first chapter of Lewis Carroll's "The Hunting of the Snark" into the images. After compression, the hidden message had a size of about 14,700 bits.

Figure 7 shows the results for the simple statistics that operate only on the redundant data. With a block size of eight bits, the result of the Maurer test for a truly random source is 7.184. We observe a noticeable increase in entropy for images that have not received statistical correction. The correlation towards one decreases noticeably, too. However, for the images that have been corrected for statistical distortions the values are very close to the data from the unmodified images. Examining only the images for which the *a priori* estimate holds, we notice that the differences between the unmodified images and the images that received corrections are even smaller. These simple tests are thus not able to detect our steganography.

The more interesting statistic is the DCT frequency histogram. If we plot the DCT histogram of images that have received corrections, we are no longer able to find noticeable differences in the distribution. Figure 8 shows the extended χ^2-test. The test detects the image without corrections and the image corrected with our naive method, but it is unable to detect the image corrected for statistical deviations with the transform in Algorithm 1.

The correcting transform causes about 2967 ± 434 additional changes to the redundant data. That is approximately 20% of the size of the hidden message. The average number of differences that can not be corrected is 186 ± 400. The majority of cor-

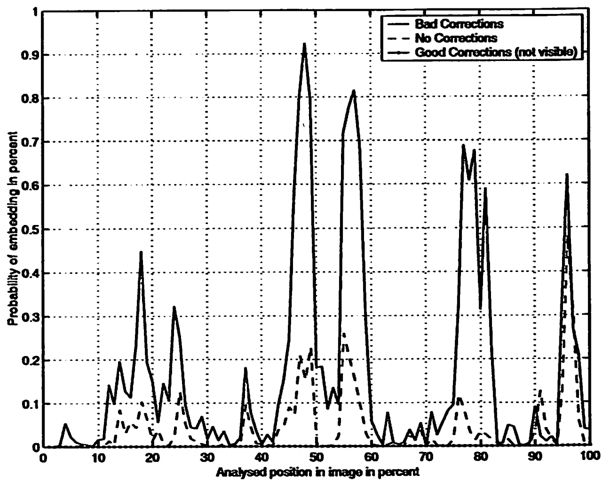


Figure 8: The extended χ^2-test detects the embedding for the image that has no statistical corrections. Our naive correction is even more detectable. However, an image that receives the proper statistical correction can not be distinguished from an unmodified image.

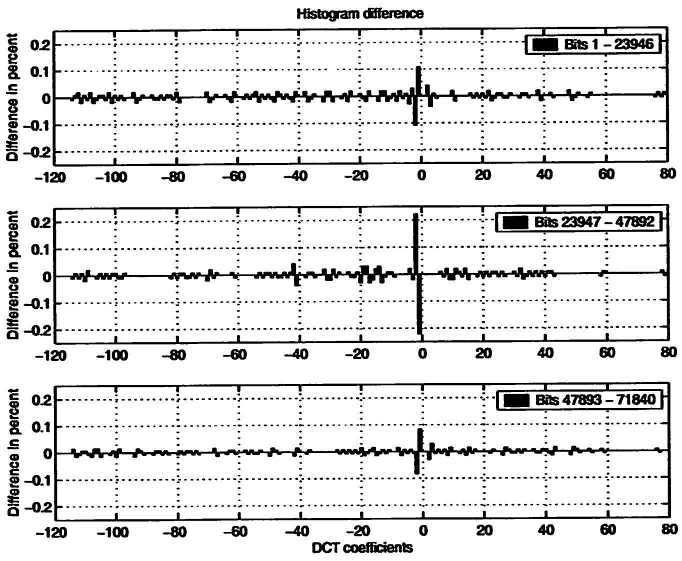


Figure 9: Examining the differences in the DCT histogram for parts of the image shows no noticeable deviations from the unmodified original. The largest difference is around 0.2%.

rections fail for images for which the *a priori* estimate indicates a maximum message size that was smaller than the one we try to embed. However, if we look only at images for which the estimate holds, the correcting transform changes 3106 ± 415 bits and the average number of differences that can not be corrected is 5 ± 5.7.

It is possible to further decrease the changes from the correcting transform, by biasing the selection algorithm against modifying coefficients with a higher frequency than their adjacent coefficients. The identification step from Section 3.1 adds the annotation that changes to bits originating from higher frequency coefficients are easier to detect. Using the worst selection, the correcting transform causes 3365 ± 442 additional changes. The best selection results in only 3054 ± 430 changes. Comparing each image, we see that using this technique, the correcting transform can avoid 311 ± 68 changes in average.

To verify the correctness of the *a priori* estimate, we embed messages of different sizes and apply the correcting transform. We note that for message sizes below the estimate the transform is able to correct most errors. Increasing the message size above the estimate causes a noticeable increase in errors.

The transform also has to meet the restriction that there be no area in the image that shows noticeable distortion in the DCT coefficients. Figure 9 shows the histogram difference of a modified image in comparison to the original. The differences in the frequency of the DCT coefficients are negligible, thus the extended χ^2-test does not indicate any hidden data.

8 Conclusion

Even though steganography is often undetectable by the observer's senses, statistical analysis can reveal the presence of a hidden message.

We introduced two new methods to improve the selection process. The first uses a seeded pseudorandom number generator to determine the fewest modifications to the cover medium. The second uses error-correcting codes to increase the flexibility in selecting bits without increasing the number of necessary changes. Together, these methods can be used to provide plausible deniability be embedding multiple hidden messages in the cover medium.

Although the commonly used χ^2-test is unable to detect modifications from the improved embedding process described in this paper, we described an extended χ^2-test that is capable of detecting modified areas in parts of an image.

To counter statistical tests based on frequency counts like the extended χ^2-test, we introduced a new method to correct the statistical deviations from the embedding process and a correcting transform for the JPEG format. As a result, none of the presented statistical tests can detect the presence of steganography. We also presented an *a priori* estimate that allows us to determine the amount of data that can be hidden in an image while still being able to preserve frequency count based statistics. Given a hidden message, we can use the estimate to quickly choose an image in which a specific message can be embedded safely.

The methods to improve the embedding and to apply correcting transforms to preserve statistical properties have been implemented in the OutGuess [8] program, which is freely available as source code at `www.outguess.org`.

9 Acknowledgments

I thank Peter Honeyman and Patrick McDaniel for careful reviews and suggestions, and Peter Gutmann for being a good shepherd. I also thank Andrew Carra for earlier discussions on reducing modifications to the cover medium, and Sean Coffey for helpful comments on error-correcting codes.

References

[1] Ross J. Anderson and Fabien A. P. Petitcolas. On The Limits of Steganography. *Journal on Selected Areas in Communication*, 16(4):474–481, May 1998.

[2] Thomas Aura. Practical Invisibility in Digital Communication. In *Proceedings of Information Hiding - First International Workshop*. Springer-Verlag, May/June 1996.

[3] Oded Goldreich. *Modern Cryptography, Probabilistic Proofs and Pseudo-randomness*. Springer-Verlag, 1999.

[4] Neil F. Johnson and Sushil Jajodia. Steganalysis of Images Created Using Current Steganographic Software. In *Proceedings of Information Hiding - Second International Workshop*. Springer-Verlag, April 1998.

[5] Ueli M. Maurer. A Universal Statistical Test for Random Bit Generators. *Journal of Cryptology*, 5(2):89–105, 1992.

[6] D. Moore and G. McCabe. *Introduction to the Pratice of Statistics*. W. H. Freeman and Company, 3rd edition, 1999.

[7] Birgit Pfitzmann. Information Hiding Terminology. In *Proceedings of Information Hiding - First International Workshop*. Springer-Verlag, May/June 1996.

[8] Niels Provos. OutGuess - Universal Steganography. `http://www.outguess.org/`, August 1998.

[9] RSA Data Security. The RC4 Encryption Algorithm, March 1992.

[10] J. H. van Lint. *Introduction to Coding Theory*. Springer-Verlag, 2nd edition, 1992.

[11] G. W. Wallace. The JPEG Still Picture Compression Standard. *Communications of the ACM*, 34(4):30–44, April 1991.

[12] Steve Walton. Information Authentication for a Slippery New Age. *Dr. Dobbs Journal*, 20(4):18–26, April 1995.

[13] Andreas Westfeld and Andreas Pfitzmann. Attacks on Steganographic Systems. In *Proceedings of Information Hiding - Third International Workshop*. Springer Verlag, September 1999.

[14] J. Zöllner, H. Federrath, H. Klimant, A. Pfitzmann, R. Piotraschke, A. Westfeld, G. Wicke, and G. Wolf. Modelling the Security of Steganographic Systems. In *Proceedings of Information Hiding - Second International Workshop*. Springer-Verlag, April 1998.

Timing Analysis of Keystrokes and Timing Attacks on SSH*

Dawn Xiaodong Song David Wagner Xuqing Tian
University of California, Berkeley

Abstract

SSH is designed to provide a secure channel between two hosts. Despite the encryption and authentication mechanisms it uses, SSH has two weakness: First, the transmitted packets are padded only to an eight-byte boundary (if a block cipher is in use), which reveals the approximate size of the original data. Second, in interactive mode, every individual keystroke that a user types is sent to the remote machine in a separate IP packet immediately after the key is pressed, which leaks the inter-keystroke timing information of users' typing. In this paper, we show how these seemingly minor weaknesses result in serious security risks.

First we show that even very simple statistical techniques suffice to reveal sensitive information such as the length of users' passwords or even root passwords. More importantly, we further show that by using more advanced statistical techniques on timing information collected from the network, the eavesdropper can learn significant information about what users type in SSH sessions. In particular, we perform a statistical study of users' typing patterns and show that these patterns reveal information about the keys typed. By developing a Hidden Markov Model and our key sequence prediction algorithm, we can predict key sequences from the inter-keystroke timings. We further develop an attacker system, *Herbivore*, which tries to learn users' passwords by monitoring SSH sessions. By collecting timing information on the network, Herbivore can speed up exhaustive search for passwords by a factor of 50. We also propose some countermeasures.

In general our results apply not only to SSH, but also to a general class of protocols for encrypting interactive traffic. We show that timing leaks open a new set of security risks, and hence caution must be taken when designing this type of protocol.

*This research was supported in part by the Defense Advanced Research Projects Agency under DARPA contract N6601-99-28913 (under supervision of the Space and Naval Warfare Systems Center San Diego) and by the National Science foundation under grants FD99-79852 and CCR-0093337.

1 Introduction

Just a few years ago, people commonly used astonishingly insecure networking applications such as `telnet`, `rlogin`, or `ftp`, which simply pass all confidential information, including users' passwords, in the clear over the network. This situation was aggravated through broadcast-based networks that were commonly used (e.g., Ethernet) which allowed a malicious user to eavesdrop on the network and to collect all communicated information [CB94, GS96].

Fortunately, many users and system administrators have become aware of this issue and have taken countermeasures. To curb eavesdroppers, security researchers designed the Secure Shell (SSH), which offers an encrypted channel between the two hosts and strong authentication of both the remote host and the user [Ylö96, SSL01, YKS+00b]. Today, SSH is quite popular, and it has largely replaced `telnet` and `rlogin`.

Many users believe that they are secure against eavesdroppers if they use SSH. Unfortunately, in this paper we show that despite state-of-the-art encryption techniques and advanced password authentication protocols [YKS+00a], SSH connections can still leak significant information about sensitive data such as users' passwords. This problem is particularly serious because it means users may have a false confidence of security when they use SSH.

In particular we identify that two seemingly minor weaknesses of SSH lead to serious security risks. First, the transmitted packets are padded only to an eight-byte boundary (if a block cipher is in use). Therefore an eavesdropper can easily learn the approximate length of the original data. Second, in interactive mode, every individual keystroke that a user types is sent to the remote machine in a separate IP packet immediately after the key is pressed (except for some meta keys such `Shift` or `Ctrl`). We show in the paper that this property can enable the eavesdropper to learn the exact length of users' passwords. More importantly, as we have verified, the time it takes the operating system to send out the packet after the key press is in general negligible comparing to the inter-keystroke timing. Hence an eavesdropper can learn the precise inter-keystroke timings of users' typing from the arrival times of packets.

Experience shows that users' typing follows stable patterns[1]. Many researchers have proposed to use the duration of key strokes and latencies between key strokes as a biometric for user authentication [GLPS80, UW85, LW88, LWU89, JG90, BSH90, MR97, RLCM98, MRW99]. A more challenging question which has not yet been addressed in the literature is whether we can use timing information about key strokes to infer the key sequences being typed. If we can, can we estimate quantitatively how many bits of information are revealed by the timing information? Experience seems to indicate that the timing information of keystrokes reveals some information about the key sequences being typed. For example, we might have all experienced that the elapsed time between typing the two letters "er" can be much smaller than between typing "qz". This observation is particularly relevant to security. Since as we show the attacker can get precise inter-keystroke timings of users' typing in a SSH session by recording the packet arrival times, if the attacker can infer what users type from the inter-keystroke timings, then he could learn what users type in a SSH session from the packet arrival times.

In this paper we study users' keyboard dynamics and show that the timing information of keystrokes does leak information about the key sequences typed. Through more detailed analysis we show that the timing information leaks about 1 bit of information about the content per keystroke pair. Because the entropy of passwords is only 4–8 bits per character, this 1 bit per keystroke pair information can reveal significant information about the content typed. In order to use inter-keystroke timings to infer keystroke sequences, we build a Hidden Markov Model and develop a n-Viterbi algorithm for the keystroke sequence inference. To evaluate the effectiveness of the attack, we further build an attacker system, Herbivore, which monitors the network and collects timing information about keystrokes of users' passwords. Herbivore then uses our key sequence prediction algorithm for password prediction. Our experiments show that, for passwords that are chosen uniformly at random with length of 7 to 8 characters, Herbivore can reduce the cost of password cracking by a factor of 50 and hence speed up exhaustive search dramatically. We also propose some countermeasures to mitigate the problem.

We emphasize that the attacks described in this paper are a general issue for any protocol that encrypts interactive traffic. For concreteness, we study primarily SSH, but these issues affect not only SSH 1 and SSH 2, but also any other protocol for encrypting typed data.

[1]In this paper we only consider users who are familiar with keyboard typing and use touch typing.

The outline of this paper is as follows. In Section 2 we discuss in more details about the vulnerabilities of SSH and various simple techniques an attacker can use to learn sensitive information such as the length of users' passwords and the inter-keystroke timings of users' passwords typed. In Section 3 we present our statistical study on users' typing patterns and show that inter-keystroke timings reveal about 1 bit of information per keystroke pair. In Section 4 we describe how we can infer key sequences using a Hidden Markov Model and a n-Viterbi algorithm. In Section 5 we describe the design, development and evaluation of an attacker system, Herbivore, which learns users' passwords by monitoring SSH sessions. We propose countermeasures to prevent these attacks in Section 7, and conclude in Section 8.

2 Eavesdropping SSH

The Secure Shell SSH [SSL01, YKS$^+$00b] is used to encrypt the communication link between a local host and a remote machine. Despite the use of strong cryptographic algorithms, SSH still leaks information in two ways:

- First, the transmitted packets are padded only to an eight-byte boundary (if a block cipher is in use), which leaks the approximate size of the original data.
- Second, in interactive mode, every individual keystroke that a user types is sent to the remote machine in a separate IP packet immediately after the key is pressed (except for some meta keys such `Shift` or `Ctrl`). Because the time it takes the operating system to send out the packet after the key press is in general negligible comparing to the inter-keystroke timing (as we have verified), this also enables an eavesdropper to learn the precise inter-keystroke timings of users' typing from the arrival times of packets.

The first weakness poses some obvious security risks. For example, when one logs into a remote site R in SSH, all the characters of the initial login password are batched up, padded to an eight-byte boundary if a block cipher is in use, encrypted, and transmitted to R. Due to the way padding is done, an eavesdropper can learn one bit of information on the initial login password, namely, whether it is at least 7 characters long or not. The second weakness can lead to some potential anonymity risks since, as many researchers have found previously, inter-keystroke timings can reveal the iden-

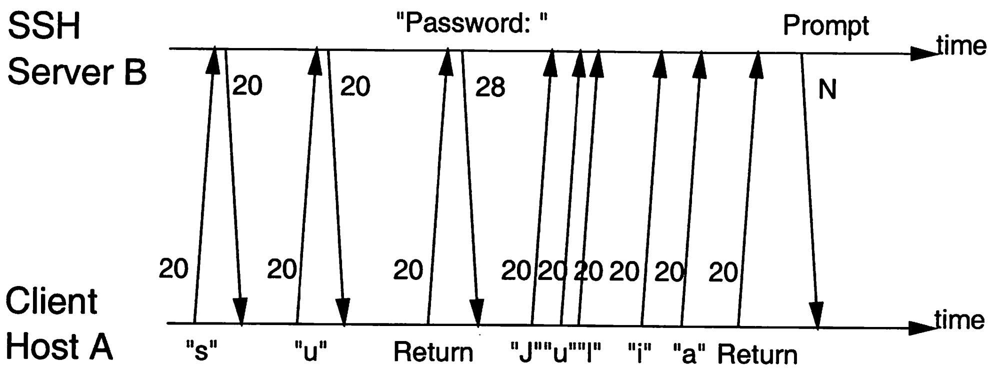


Figure 1: The traffic signature associated with running SU in a SSH session. The numbers in the figure are the size (in bytes) of the corresponding packet payloads.

tity of the user [GLPS80, UW85, LW88, LWU89, JG90, BSH90, MR97, RLCM98, MRW99].

In this section, we show that several simple and practical attacks exploiting these two weaknesses. In particular, an attacker can identify which transmitted packets correspond to keystrokes of sensitive data such as passwords in a SSH session. Using this information, the attacker can easily find out the exact length of users' passwords and even the precise inter-keystroke timings of the typed passwords. Learning the exact length of users' passwords allows eavesdroppers to target users with short passwords. Learning the inter-keystroke timing information of the typed passwords allows eavesdroppers to infer the content of the passwords as we will show in Section 3 and 4.

Traffic Signature Attack We can often exploit properties of applications to identify which packets correspond to the typing of a password. Consider, for instance, the SU command. Assume the user has already established a SSH connection from local host *A* to remote host *B*. When the user types the command SU in the established SSH connection $A \leftrightarrow B$, we obtain a peculiar traffic signature as shown in Figure 1. If the SSH session uses SSH 1.x[2] and a block cipher such as DES for the encryption [NBS77, NIS99], as is common, then the local host *A* sends three 20-byte packets: "s", "u", "Return". The remote host *B* echoes the "s" and "u" in two 20-byte packets and sends a 28-byte packet for the "Password: " prompt. Then *A* sends 20-byte packets, one for each of the password characters, without receiving any echo data packets. *B* then sends some final packets containing the root prompt if SU succeeds, otherwise some failure messages. Thus by checking the traffic against this "su" signature, the attacker can identify when the user issues the SU command and hence learn which packets correspond to the password keystrokes. Note that similar techniques can be used to identify when users type passwords to authenticate to other applications such as PGP [Zim95] in a SSH session.

[2]The attack also works when ssh 2.x is in use. Only the packet sizes are slightly different.

Multi-User Attack Even more powerful attacks exist when the attacker also has an account on the remote machine where the user is logging into through SSH. For example, the process status command ps can list all the processes running on a system. This allows the attacker to observe each command that any user is running. Again, if the user is running any command that requires a password input (such as su or pgp) the attacker can identify the packets corresponding to the password keystrokes.

Nested SSH Attack Assume the user has already established a SSH session between the local host *A* and remote host *B*. Then the user wants to open another SSH session from *B* to another remote host *C* as shown in Figure 2. In this case, the user's password for *C* is transmitted, one keystroke at a time, across the SSH-encrypted link $A \leftrightarrow B$ from the user to *B*, even though the SSH client on machine *B* patiently waits for all characters of the password before it sends them all in one packet to host C for authentication (as designed in the SSH protocol [YKS+00a]). It is easy to identify such a nested SSH connection using techniques developed by Zhang and Paxson [ZP00b, ZP00a]. Hence in this case the eavesdropper can easily identify the packets corresponding to the user's password on link $A \leftrightarrow B$, and from this learn the length and the inter-keystroke timings of the users' password on host *C*.

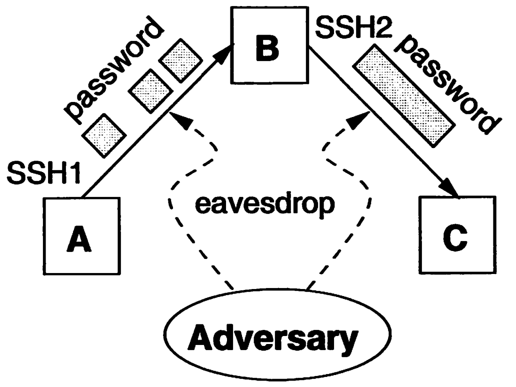


Figure 2: The nested SSH attack.

3 Statistical Analysis of Inter-keystroke Timings

As a first study towards inferring key sequences from timing information, we develop techniques for statistical analysis of the inter-keystroke timings. In this section, we first describe how we collect training data and show some simple timing characteristics of character pairs. We then show how we model the inter-keystroke timing of a given character pair as a Gaussian distribution. We then describe how to estimate quantitatively the amount of information about the character pair that one can learn using the inter-keystroke timing information. Denote the set of character pairs of interest as Q, and let $|Q|$ denote the cardinality of the set Q.

3.1 Data Collection

The two keystrokes of a pair of characters (k_a, k_b) generates four events: the press of k_a, the release of k_a, the press of k_b, and the release of k_b. However, because only key presses (not key releases) trigger packet transmission, an eavesdropper can only learn timing information about the key-press events. Since the main focus of our study is in the scenario where an adversary learns timing information on keystrokes by simply monitoring the network, we focus only on key-press events. The time difference between two key presses is called the *latency* between the two keystrokes. We also use the term *inter-keystroke timing* to refer to the latency between two keystrokes.

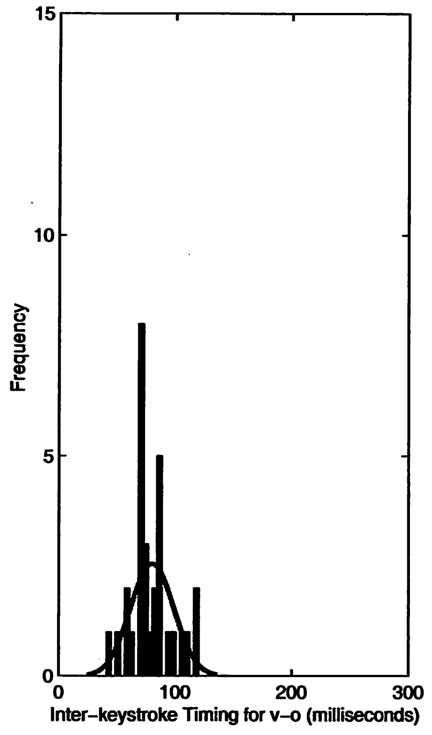
In order to characterize how much information is leaked by inter-keystroke timings, we have performed a number of empirical tests to measure the typing patterns of real users. Because passwords are probably the most sensitive data that a user will ever type, we focus only on information revealed about passwords (rather than other forms of interactive traffic).

Our focus on passwords creates many challenges. Passwords are entered very differently from other text: passwords are typed frequently enough that, for many users, the keystroke pattern is memorized and often typed almost without conscious thought. Furthermore, well-chosen passwords should be random and have little or no structure (for instance, they should not be based on dictionary words). As a consequence, naive measurements of keystroke timings will not be representative of how users type passwords unless great care is taken in the design of the experimental methodology.

Our experimental methodology is carefully designed to address these issues. Due to security and privacy considerations, we chose not to gather data on real passwords; therefore, we have chosen a data collection procedure intended to mimic how users type real passwords. A conservative method is to pick a random password for the user (where each character of the password is chosen uniformly at random from a set of 10 letter keys and 5 number keys, independently of all other characters in the password), have the user practice typing this password many times without collecting any measurements, and then measure inter-keystroke timing information on this password once the user has had a chance to practice it at length.

However, we found that, when the goal is to try to identify potentially relevant timing properties (rather than verify conjectured properties), this conservative approach is inefficient. In particular, users typically type passwords in groups of 3–4 characters, with fairly long pauses between each group. This distorts the digraph statistics for the pair of characters that spans the group boundary and artificially inflates the variance of our measurements. As a result we would need to collect a great deal of data for many random passwords before this effect would average out. In addition, it takes quite a while for users to become familiar with long random passwords. This makes the conservative approach a rather blunt tool for understanding inter-keystroke statistics.

Fortunately, there is a less costly way to gather inter-keystroke timing statistics: we gather training data on each pair of characters (k_a, k_b) as typed in isolation. We pick a character pair and ask the user to type this pair 30–40 times, returning to the home row each time between repetitions. For each user, we repeat this for many possible pairs (142 pairs, in our experiments) and we gather data on inter-keystroke timings for each such pair. We collected the latency of each character pair measurement and computed the mean value and the standard deviation. In our experience, this gives better results.



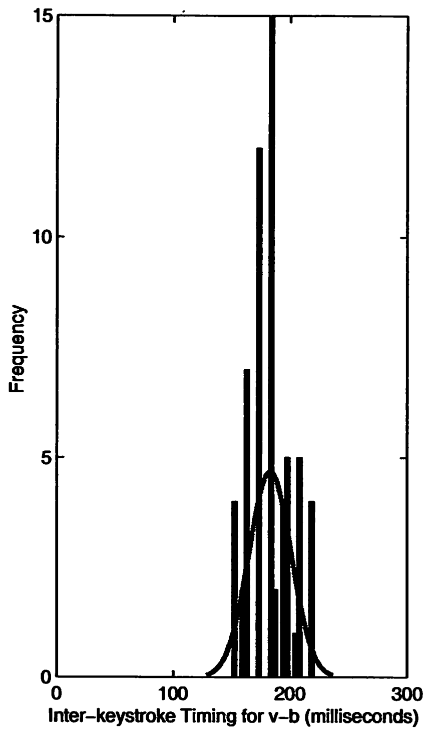


Figure 3: The distribution of inter-keystroke timings for two sample character pairs.

As an example, Figure 3 shows the latency histogram of two sample character pairs. The left model corresponds to the latency between the pair (v, o), and the right model corresponds to (v, b). We can see that the latency between (v, o) is clearly shorter than the latency between (v, b), and the latency distributions of these two sample character pairs are almost entirely non-overlapping.

The optimized data collection approach gives us a more efficient way to study fine-grained details of inter-keystroke statistics without requiring collecting an enormous amount of data. We used data collected in this way to quickly identify plausible conjectures, develop potential attacks, and to train our attack models. As far as we are aware, collecting data on keystroke pairs in isolation does not seem to bias the data in any obvious way. Nonetheless, we also validate all our results using the conservative measurement method (see Section 5).

3.2 Simple Timing Characteristics

Next, we divide the test character pairs into five categories, based on whether they are typed using the same hand, the same finger, and whether they involve a number key:

- Two letter keys typed with alternating hands, i.e., one with left hand and one with right hand;
- Two characters containing one letter key and one number key typed with alternating hands;
- Two letter keys, both typed with the same hand but with two different fingers;
- Two letter keys typed with the same finger of the same hand;
- Two characters containing one letter key and one number key, both typed with the same hand.

Figure 4 shows the histogram of latency distribution of character pairs for each category. We split the whole latency range into six bins as shown in the x-axis. Within each category, we put each character pair into the corresponding bin if its mean latency value is within the range of the bin. Each bar in the histogram of a category represents the ratio of the number of character pairs in the associated bin over the total number of character pairs in the category.[3] We can see that all the character pairs that are typed using two different hands take less than 150 milliseconds, while pairs typed using the same hand and particularly the same finger take substantially longer. Character pairs that alternate between one letter key and one number key, but are typed using the same

[3]Hence the sum of all bars within one category is 1.

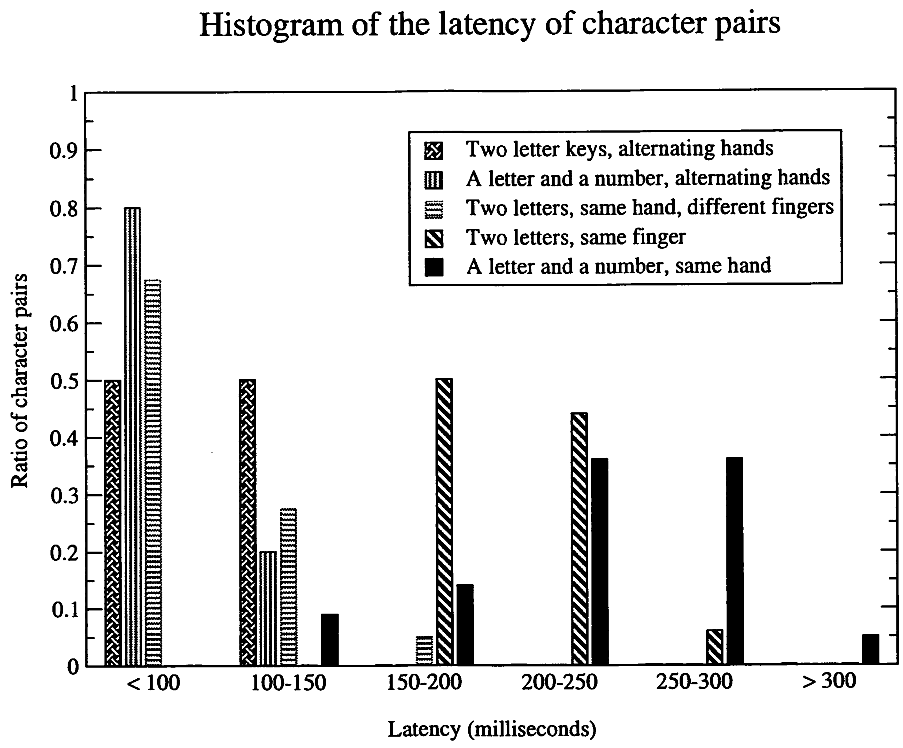


Figure 4: Inter-keystroke timings for character pairs in five different categories. Note that some bars at some positions disappear because the corresponding height is zero.

hand, take the longest time to type. This is simply because two hands offer a certain amount of parallelism, while character pairs typed with one hand require a certain degree of sequential movements and hence tend to take longer. This is especially obvious in the case of one letter and one number pairs typed using one hand. They in general require more hand movement and hence the longest time.[4]

So, if the attacker observes a character pair typed with latency more than 150 milliseconds, he can guess with high probability of success that the character pair is not typed using two different hands and hence can infer about 1 bit of information about the content of the character pair. Because the 142 character pairs are formed from randomly selected letter keys and number keys, they seem likely to form a representative sample of the whole keyboard. Hence this simple classification extends to the whole keyboard, and already indicates that the inter-keystroke timing leaks substantial information about what is typed.

The properties described above are unlikely to be exhaustive. For instance, earlier work on timing attacks on multi-user machines suggested that inter-keystroke timings may additionally reveal which characters in the password are upper-case [Tro98].

[4]Note that here we only consider users that use touch typing.

3.3 Gaussian Modeling

From the plot of the latency distribution of a given character pair, such as the ones shown in Figure 3, we can see that the latency between the two key strokes of a given character pair forms a Gaussian-like unimodal distribution. Hence a natural assumption (which is confirmed by our empirical observations) is that the probability of the latency y between two keystrokes of a character pair $q \in Q$, $\Pr[y|q]$, forms a univariate Gaussian distribution $\mathcal{N}(\mu_q, \sigma_q)$, meaning

$$\Pr[y|q] = \frac{1}{\sqrt{2\pi}\sigma_q} e^{-\frac{(y-\mu_q)^2}{2\sigma_q^2}},$$

where μ_q is the mean value of the latency for character pair q and σ_q is the standard deviation. Given a set of training data $\{(q_i, y_i)\}_{1 \le i \le N}$, where q_i is the i-th character pair and y_i is the corresponding latency in the data collection, we can derive the parameters $\{(\mu_q, \sigma_q)\}_{q \in Q}$ based on *maximum likelihood* estimation, i.e., we compute the mean and the standard deviation for each character pair.

Figure 5 shows the estimated Gaussian models of the latencies of the 142 character pairs. Our empirical result

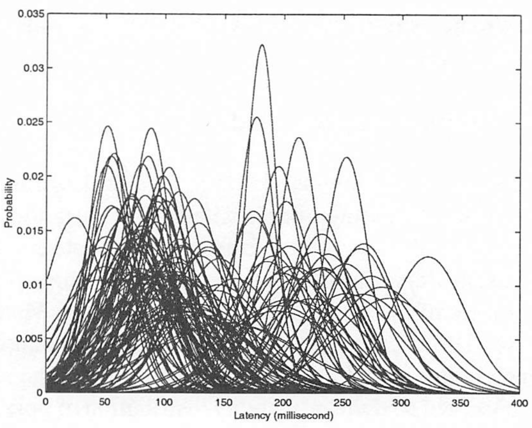
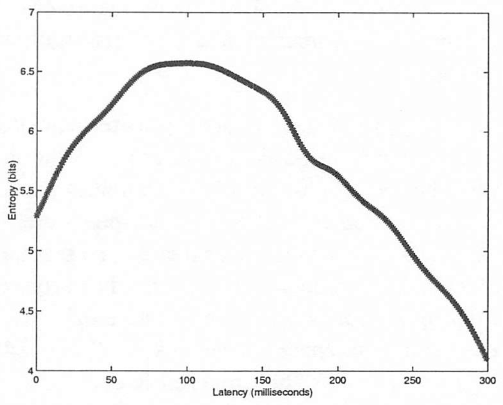
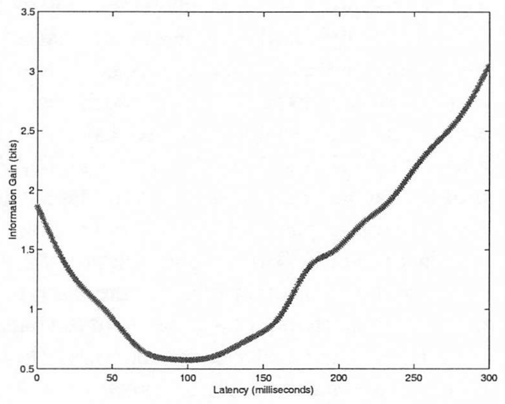


Figure 5: Estimated Gaussian distributions of all 142 character pairs collected from a user.



(a) Entropy of character pairs given a latency observation



(b) Information gain induced by a latency observation

Figure 6: Entropy and information gain as a function of the inter-keystroke latency.

shows that most of the latencies of the character pairs lie between 50 and 250 milliseconds. The average of the standard deviation of the 142 character pairs is about 30 milliseconds. The graph also indicates that the latency distributions of the character pairs severely overlap, which means the inference of character pairs using just latency information is a challenging task.

3.4 Information Gain Estimation

We would like to estimate quantitatively how much information the latency information reveals about the character pairs typed. This will be an upper bound of how much information an attacker can extract from the timing information using any particular method. We estimate it by computing the *information gain* induced by the latency information. If we select a character pair uniformly at random from the character-pair space, and if the attacker does not get any additional information, the entropy of the probability distribution of character pairs to the attacker is $H_0[q] = -\sum_{q \in Q} \Pr[q] \log_2 \Pr[q] = \log_2 |Q|$. If the attacker learns the latency y_0 between the two keystrokes of the character pair, the estimated entropy of the probability distribution of character pairs to the attacker is $H_1[q|y = y_0] = -\sum_{q \in Q} \Pr[q|y_0] \log_2 \Pr[q|y_0]$, where $\Pr[q|y_0] = \frac{\Pr[y_0|q] \cdot \Pr[q]}{\sum_{q \in Q} \Pr[y_0|q] \cdot \Pr[q]}$, and $\Pr[y_0|q]$ is computed using the Gaussian distribution obtained in the parameter estimation phase in the previous subsection. The information gain induced by the observation of latency y_0 is the difference between the two entropies, $H_0[q] - H_1[q|y = y_0]$. Using the parameter estimation of the 142 character pairs obtained in the previous section, we can compute $H_1[q|y = y_0]$ and $H_0[q] - H_1[q|y = y_0]$ as shown in Figure 6(a) and Figure 6(b).

The estimated information gain, also called *mutual information*, is $\mathrm{I}[q; y] = H_0[q] - H_1[q|y] = H_0[q] - \int \Pr[y_0] \cdot H_1[q|y = y_0] dy_0$, where $\Pr[y_0] = \sum_{q \in Q} \Pr[y_0|q] \Pr[q]$. From the numerical computation we obtain $\mathrm{I}[q; y] = 1.2$. This means the estimated information gain available from latency information is about 1.2 bits per character pair when the character pair has uniform distribution. Hence the attacker could potentially extract 1.2 bits of information per character pair by using the latency information in this case. Because the character pairs in our experiments are selected uniformly at random from all letter and number keys, we expect that they will be representative of the whole keyboard. Intuitively, Figure 5 is a sufficiently-large random sampling of a much denser graph containing the latency distributions of all possible character pairs. More detailed analysis shows that the estimated information gain computed using 142 sample character pairs is a good estimate of the information gain when the character-pair space includes all letter and number character pairs. This estimate is comparable to the back-of-the-envelope calculation in Section 3.2 based on our classification into five categories of keystroke pairs.

Because the entropy of written English is so low (about 0.6–1.3 bits per character [Sha50]), the 1.2-bit information gain per character pair leaked through the latency information seems to be significant.[5] For example, we can expect that users' PGP passphrases will often contain only 1 bit of entropy per character. Hence the latency information may reveal significant information about PGP passphrases.

The information gain curve in Figure 6(b) shows a convex shape. Note that latencies greater than 175 milliseconds are relatively rare; however, whenever we see such a long time between keystrokes, we learn a lot of information about what was typed, because there are not many possibilities that would lead to such a large latency. The character pairs that take longer than 175 milliseconds to type are mostly pairs containing number keys or pairs typed with one finger. Hence this analysis suggests that passwords containing number keys or character pairs that are typed with one finger are particularly vulnerable to such timing attacks.

Another interesting observation is that the mean of the standard deviations of the character pairs is only about 30 milliseconds as shown in our experiments, while the standard deviation of round-trip time on the Internet in many cases is less than 10 milliseconds [Bel93]. Therefore even when the attacker is far from the SSH client host, he can still get sufficiently-precise inter-keystroke timing information. This makes the timing attack even more severe.

4 Inferring Character Sequences From Inter-Keystroke Timing Information

In this section, we describe how we can infer character sequences using the latency information. In particular, we model the relationship of latencies and character sequences as a Hidden Markov Model [RN95]. We extend the standard Viterbi algorithm to an n-Viterbi algorithm that outputs the n most likely candidate character sequences. We further estimate how many bits of information about the real character sequence this algorithm extracts from the latency information and show it is nearly optimal.

4.1 Hidden Markov Model

In general, a Markov Model is a way of describing a finite-state stochastic process with the property that the probability of transitioning from the current state to another state depends only on the current state, not on any prior state of the process [RN95]. In a Hidden Markov Model (HMM), the current state of the process cannot be directly observed. Instead, some outputs from the state are observed, and the probability distribution of possible outputs given the state is dependent only on the state. Using a HMM, one can infer information about the prior path the process has taken from the sequence of observed outputs of the states, and efficient algorithms are known for working with HMM's. Because of this, HMM's have been widely used in areas such as speech recognition and text modeling.

In our setting, we consider each character pair of interest as a hidden (non-observable) state, and the latency between the two keystrokes of the character pair as the output observation from the character-pair state. Each state corresponds to a pair of characters, so that the typing of a character sequence $K_0, \ldots, K_T$, is a process that goes through T states, $q_1, \ldots, q_T$, where $q_t (1 \leq t \leq T)$ represents the t-th character pair (K_{t-1}, K_t) typed. Let $y_t (1 \leq t \leq T)$ denote the observed latency of state q_t. Then we model the typing of a character sequence as a HMM. This means we make two assumptions. First, the probability of transition from the current state to another state is only dependent on the current state, not on the prior path of the process. If the character sequence is a password chosen uniformly at random, this assumption obviously holds. In the case of text, this assumption does not hold strictly but experience in speech recognition and text modeling shows that some extensions to HMM still work well [RN95]. Second, the probability distribution of the latency observation is only dependent on the current character pair and not on any previous characters in the sequence. This assumption might hold for some cases and not for other cases where the typing of previous characters changes the position of the hand and influences the typing of later character pairs. However, making this assumption makes our analysis and inference algorithm much simpler and still gives good results as shown from the experiments. Hence, we use a HMM to model the typing of character sequences as shown in Figure 7.

As in the previous section, we assume the set of possible character pairs is Q, hence the set of possible states in the HMM is Q. We assume that the probability of

[5] Note that the 1.2-bit information gain is estimated for the case of randomly selected passwords where the sequence of characters have a uniform distribution. However, this is not the case for texts. More careful calculation is needed to estimate the information gain in the case of natual text.

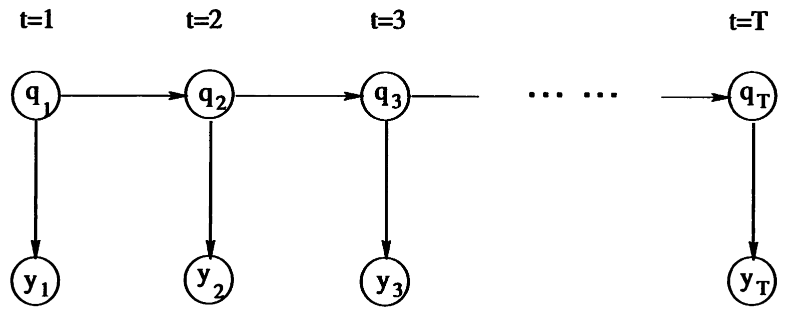


Figure 7: A representation of a trace of a HMM. Each vertical slice represents a time step. In each time slice, the top node q_t is a variable representing a character pair, and the bottom node y_t is the observable variable denoting the latency between the two keystrokes.

the latency y of a character pair q, $\Pr[y|q]$ ($q \in Q$), is a Gaussian distribution $\mathcal{N}(\mu_q, \sigma_q)$, where the parameters $\{(\mu_q, \sigma_q)\}_{q \in Q}$ are obtained using the maximum likelihood estimation.

4.2 The n-Viterbi Algorithm for Character Sequence Inference

Given an observation $\vec{y} = (y_1, y_2, \dots, y_T)$, a sequence of latencies of some character sequence from a user's typing, we would like to infer the real character sequence that the user has typed. For each possible character sequence $\vec{q} = (q_1, q_2, \dots, q_T)$, we can compute how likely the character sequence is given the observation, namely $\Pr[\vec{q}\,|\vec{y}]$. The probability $\Pr[\vec{q}\,|\vec{y}]$ essentially gives a ranking for the candidate character sequence $\vec{q}$: the higher $\Pr[\vec{q}\,|\vec{y}]$ is, the more likely $\vec{q}$ is the real character sequence. We use $\vec{q}^{\,*}$ to denote the most-likely sequence, which is the sequence that corresponds to the highest value of $\Pr[\vec{q}|\vec{y}]$ for all possible $\vec{q}$ with regard to a given $\vec{y}$.

The Viterbi algorithm is widely used in solving the most likely sequence of states given a sequence of observation in HMM problems [RN95]. An naive way of computing $\vec{q}^{\,*}$ would compute $\Pr[\vec{q}|\vec{y}]$ for all possible $\vec{q}$, and hence requires $O(|Q|^T)$ running time. The Viterbi algorithm uses dynamic programming for a running time complexity $O(|Q|^2T)$.

In our setting, because the latency distributions of different character pairs highly overlap, the probability that the most likely sequence is the right sequence will be very low. Hence, instead of just computing the most likely sequence, we need to compute the n most likely sequences and hope the real sequence will be in the n most likely sequences with high probability for n greater than a certain threshold. Hence we extend the standard Viterbi algorithm to n-Viterbi algorithm to output the n most-likely sequences with running time complexity $O(n|Q|^2T)$. We give a detailed description of the n-Viterbi algorithm in Appendix A.

4.3 How to Estimate the Effectiveness of the n-Viterbi Algorithm

We would like to estimate how big the threshold n has to be such that the real character sequence will be among the n most-likely sequences with sufficiently high probability. In an experiment if the real character sequence appears in the n most-likely sequences, we say the experiment is a success with regard to the threshold n, otherwise, a failure. The probability of such defined success is a function of n. It is easy to see that the function is monotonically increasing with regard to n. If for a small n, the success probability is already high, this means the algorithm is very effective because it filters out most of the sequences and hence one only needs to try a small set of candidates before finding the real sequence. On the other hand, if we need a high threshold of n to get a sufficiently high success probability, then the algorithm is less effective: one would need to try many more candidates before finding the real sequence. Note that from Section 3.4 we see that the timing information reveals about 1.2 bits of information per character pair. For the case of a random password of length $T+1$, which forms T consecutive character pairs, the latency information could reveal approximately $1.2T$ bits of information about the real password sequence. Hence this is an upper bound on the effectiveness of the algorithm to infer character sequences using latency information. We would like to estimate how close our algorithm is compared to the upper bound.

First, we look at the simple case when $T = 1$. Given a latency observation y of a character pair q, we compute the probability $\Pr[q'|y], q' \in Q$, and select the n most-likely character pairs $\Phi = \{q_{j_1}, \dots, q_{j_n}\}$. We would like to compute the probability that the real character pair q is in the set Φ over all possible values of y. To simplify

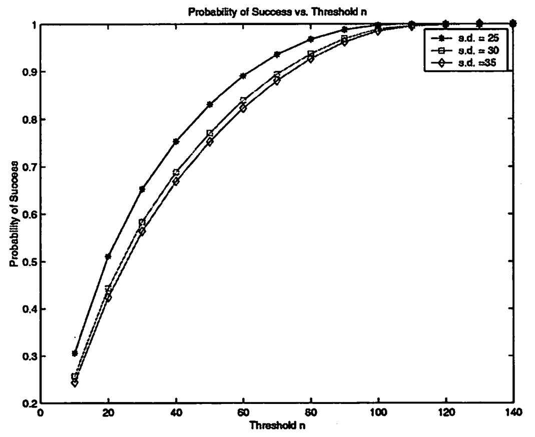


Figure 8: The probability that the n-Viterbi algorithm outputs the correct password before the first n guesses, graphed as a function of n.

the numerical computation, we approximate the result by assuming that all the Gaussian distributions have the same standard deviation σ. This is a good approximation of the real experiment: as we see in the Figure 5, most keypairs have a standard deviation between 25–35 milliseconds.

Figure 8 graphs the probability that the real character pair appears within the n most-likely character pairs against the threshold n. The top curve is when $\sigma = 25$, the middle curve is when $\sigma = 30$, and the bottom curve is when $\sigma = 35$. Using the middle curve, we get that when $n = 70$ the probability of success is 90%, meaning that with 90% probability, the real character pair appears in the 70 most-likely sequences output by the n-Viterbi algorithm. Let's denote such a threshold corresponding to the 90% success probability as n^*. Thus $\log_2(|Q|/n^*) = 1$ is the approximate number of bits of information per character pair the algorithm extracts. Note that from the previous section we see that the latency information reveals about 1.2 bits of information per character pair. Hence our n-Viterbi algorithm is near-optimal.

In the case of uniformly randomly chosen passwords of length $T + 1$, the number of bits of information the algorithm can extract is approximately $T \cdot \log_2(|Q|/n^*) \approx T$, which is close to the optimal value $1.2T$ bits.

5 Building Herbivore and Timing Attacks on SSH

To evaluate the effectiveness of our timing attacks to SSH, we build an attacker program that we call *Herbivore*. In this section, we describe the experiment results of using Herbivore to learn users' passwords.

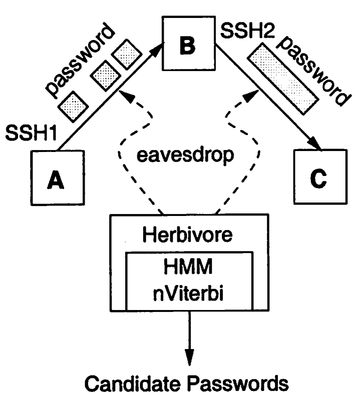


Figure 9: The Herbivore architecture.

5.1 Herbivore Preying for Passwords

We built an attacker engine Herbivore as shown in Figure 9. It monitors the network and collects the arrival times of packets. Using the technique described in Section 2, Herbivore infers which packets correspond to the user's SSH passwords when the user opens an SSH session to another host within an established SSH connection. Herbivore then measures the inter-arrival times between packets containing the password characters and uses our n-Viterbi algorithm to generate a list of candidate passwords. The candidate passwords are sorted in decreasing order of the probability $\Pr[\vec{q}\ |y]$, and in our experiments we record the position of the real password in the candidate list. We report the position of the password as a percentage, so with m possible passwords in total, if the real password appears at position u in the ordered candidate list, we say the real password appears at the top $\frac{100u}{m}\%$. This gives a natural way to quantify the effectiveness of our approach.

5.2 Optimization for Long Character Sequences

The complexity of the n-Viterbi algorithm is linear in the number n of candidates it outputs. As the length of the password grows, the space of possible passwords grows exponentially. If the n-Viterbi algorithm can only rule out a constant fraction of the password space, n would also grow exponentially as the password length grows. Hence the algorithm might be inefficient when the password is long. In particular, we observed that memory usage can grow substantially for longer passwords.

Also, and more importantly, we observed in the experiments that users tend to type long passwords in segments of 3 to 5 letters and pause between the segments. If we

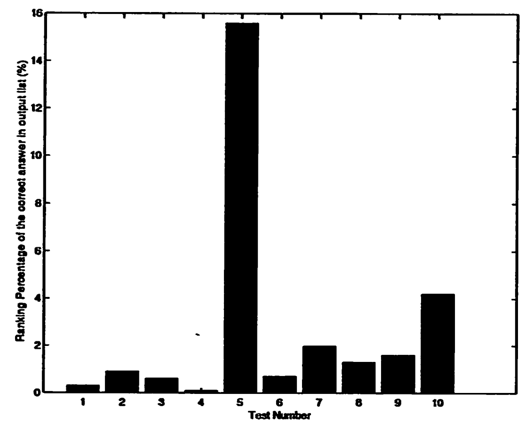


Figure 10: The percentage of the password space tried by Herbivore in 10 tests before finding the right password.

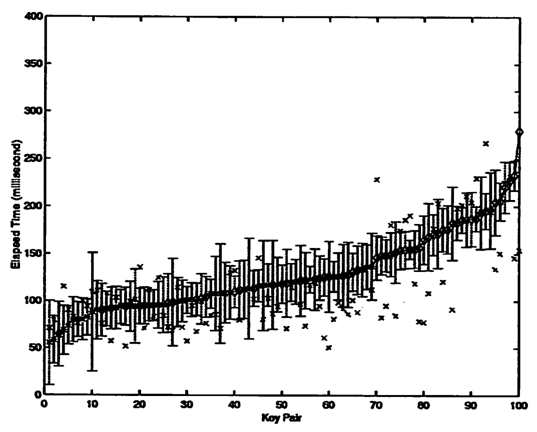


Figure 11: A comparison of two users' typing patterns. The "diamond" symbols show the mean values of the latencies of one user, with an error-bar indicating one standard deviation. The "x" symbol indicates the mean values of the latencies of another user.

use the timing between the segments for the prediction, it might bias our predictions since typically such pauses are noticeably longer than most other inter-keystroke latencies. Fortunately, this large difference means that pauses between groups of password characters can be clearly identified before we apply the *n*-Viterbi algorithm.

Hence to reduce the bias and to reduce the memory requirements of the algorithm, we break the timing information of the password into segments containing 3 or 4 latency intervals. We use each segment to form a HMM and then at the end combine the result from different segments to form the candidate password ordering.

5.3 Experimental Results for Password Inference for a Single User

We measure the effectiveness of our *n*-Viterbi algorithm at cracking passwords through empirical measurements. In our experiment, we use training data compiled from isolated keypairs to train the HMM. Then, we pick a random password for the user. We have the user use this password to authenticate to another SSH session within an established SSH session as shown in Figure 9, and we apply our *n*-Viterbi algorithm to simulate an attack on this password. Note that we have the test user type the password many times before the test to ensure familiarity with the password, and we try to deduce the user's password using training data from the same user.

All passwords are selected uniformly at random from the character space as in the experiment in Section 3, so they contain no structure. Recovering such passwords is the hardest case for the attacker, so if timing analysis can recover information in such a scenario, we can expect that timing analysis will be an even greater threat in settings where passwords are chosen less carefully.

We performed tests for 10 different passwords, each of length 8. Figure 10 shows the percentage of the positions of the real password in the ordered candidate lists output by the *n*-Viterbi algorithm. For example, 0.3% means that the real password appeared at the top 0.3% position in the output candidate list. These experiments indicate that on average the real password is located within the top 2.7% of the candidate ranking list. The median position is about 1%, so about half the time the password will be in the top 1% of the list of candidates produced by our *n*-Viterbi algorithm. Therefore, in order to crack the password, Herbivore only needs to test $1/50$ times as many passwords as brute-force search, on average.

The $50\times$ reduction in workfactor compared to exhaustive search corresponds to a total of 5.7 bits of information learned per password using the latency information. This is close to the information gain analysis in Sections 3 and 4, which predicted a gain of about 1 bit per keystroke pair: recall that the passwords in this test are of length 8, so each password contains 7 keystroke pairs. We attribute the difference to minor variation between the distributions of inter-keystroke timings in random passwords and the distribution of timings for character pairs typed in isolation.

For ease of testing, our experiments were on passwords with a reduced set of possible characters. However, we can expect these results to carry over to passwords chosen from the full set of possible characters. Assuming that the information gain available from inter-keystroke timing information is about 1 bit per character pair even

Training Set	Test Set	Test Cases				
		Password 1	Password 2	Password 3	Password 4	Password 5
User 1	User 1	15.6%	0.7%	2.0%	1.3%	1.6%
User 1	User 2	62.3%	15.2%	7.0%	14.8%	0.3%
User 1	User 3	6.4%	N/A	1.8%	3.1%	4.2%
User 1	User 4	1.9%	31.4%	1.1%	0.1%	28.8%
User 2	User 1	4.9%	1.3%	1.6%	12.3%	3.1%
User 2	User 2	30.8%	15.0%	2.8%	3.7%	2.9%
User 2	User 3	4.7%	N/A	5.3%	6.7%	38.4%
User 2	User 4	0.7%	16.8%	3.9%	0.6%	5.4%

Table 1: Success rates for password inference with multiple users. The numbers are the percentage of the search space the attacker has to search before he finds the right password.

when we extend to the whole keyboard, we expect to see this 50 times reduction in workfactor for passwords of length 7–8 even when the passwords are chosen randomly from all letter and number keys. This 50× reduction can make password cracking more practical. For example, for a password containing randomly-selected lower-case letter keys and number keys, without timing information, the attacker would need to try $36^8/2$ candidate passwords on average before he finds the right one. Benchmarks indicate that a 840 MHz Pentium III can check about 250,000 candidate passwords per second in a off-line dictionary attack. Thus, exhaustive search would take about 65 PC-days to crack a password composed of randomly-selected lower-case letter keys and number keys. If the attacker uses the timing information, the computation can be done in 1.3 days, which makes the crack 50× more practical.

5.4 Experimental Results for Password Inference for Multiple Users

One potential weakness in our simulations is that real-world attackers might not be able to get as much training data from the victim for the statistical analysis as we had available in our experiments. However, we argue next that this is unlikely to pose an effective defense against timing attacks: there are other ways that attackers can obtain the training data required for the attack.

One simple observation is that the attacker can easily get his own typing statistics, or the typing statistics of a co-conspirator. Hence it is important to evaluate how well the password inference techniques perform when using one person's typing statistics to infer passwords typed by another person.

In this experiment, we collected the typing statistics of two users, User 1 and User 2. An interesting result is that 75% of the character pairs take about the same latency to type for both two users: in other words, the difference between the average latencies of the two users for such character pairs is smaller than one standard deviation. Similarly, the simple timing characteristics reported in Section 3.2—e.g., keypairs typed with alternate pairs tend to have much lower inter-keystroke latency than keypairs typed with the same hand—were observed to be essentially user-independent. This suggests that typing statistics have a large component that is common across a broad user population and which thus can be exploited by attackers even in the absence of any training data from the victim.

To test this hypothesis further, we had four users (including User 1 and 2, from our previous experiments) type the same set of five randomly-selected passwords. Passwords 1 and 2 have length 8. Passwords 3 and 4 have length 7, and password 5 has length 6. Herbivore then runs the n-Viterbi algorithm using the typing statistics from User 1 and 2 to infer passwords typed by the four test users separately. Table 1 shows the percentage position of the real passwords occurred in the output candidate ranking list, which is the percentage of the password space the attacker has to search before he finds the right password. User 3 did not type Password 2 so the entry is not available.

This experiment shows several interesting results:

- Unsurprisingly, inferring a user's password can in general be done somewhat more effectively if one uses training data from the same user rather than training data from other users.
- The distance between the typing statistics of two users can vary significantly according to how one chooses the pair of users. A user U_a's typing pattern might be more similar to user U_b's than to user U_c's. Thus it can give better results to use U_b's training data than U_c's training data to infer passwords typed by U_a. In this experiment, it shows

that in general using User 1's training data gives a better result to infer passwords typed by User 3 than using User 2's training data. And User 2's training data gives a better inference for passwords typed by User 4 than User 1's training data.

- Most importantly, this experiment shows that training data from one user can be successfully applied to infer passwords typed by another user. Hence the attack can be effective even when the attacker does not have typing statistics from the victim.

5.5 Extensions

We expect that Herbivore could also be used to infer information about text or commands that users type. The entropy of written English is very low (about 0.6–1.3 bits per character [Sha50]) in comparison to the amount of information leaked by inter-keystroke timings (about 1 bit of information per key pair; see Section 3). However, mounting such an attack would appear to require better models of written text [RN95]. In any case, we have not studied such a scenario in our experiments, and we leave this for future work.

6 Related Work

Timing analysis has previously been used by Kocher to attack cryptosystems [Koc95]. Trostle exploited a similar idea, showing how a malicious user on a multiuser workstation can gain information about other users' passwords using CPU timings [Tro98]. We expect our Hidden Markov Model techniques might find applications in Trostle's threat model as well.

Most recently, other researchers have independently pointed out the possibility of timing attacks on SSH [DS01]. Some of their observations reveal additional weaknesses in SSH: For instance, they noted that the SSH 1.x protocol reveals the exact length of passwords, because ciphertexts contain a length field sent in the clear (SSH 2 does not have this problem); they discussed how to deal with the presence of backspace characters; and, they initiated an investigation of the impact of timing attacks on other session data (such as shell commands typed in the SSH session).

7 Countermeasures

Although SSH provides an encrypted and authenticated link between the local host and the remote machine, an eavesdropper can still learn information about typed keystrokes due to two weaknesses of SSH. First, every individual keystroke that a user types is sent to the remote machine in an individual IP packet (except for meta keys such as `Shift` and `Ctrl`); second, as soon as command output is available on the remote machine, it is sent to the local host in one or multiple IP packets, leaking information on the approximate size of the output. We have shown in this paper how these seemingly minor weaknesses lead to severe real-world attacks.

Note that in our traffic signature attack, the attacker can tell that the user is typing passwords because there are no echo packets. So one way to fix this problem is that when the server detects that the echo mode is turned off, the server can return dummy packets that will be ignored by the client when it receives keystroke packets from the client. This fix can reduce the effectiveness of the traffic signature attack but could fail in other attacks such as our nested SSH attack where the attacker can guess when the user is typing his password by simply monitoring the network connections. This fix does not prevent inter-keystroke timing information, though.

To prevent the attacks, we need to prevent the leakage of the timing information of the keystrokes. One naive approach might be to modify SSH so that upon receiving a keystroke with latency less than η milliseconds from the previous keystroke, the program will delay the packet by a random amount of up to η milliseconds. Because our experiment indicates that the spectrum of the latency between two keystrokes of continuous typing is between 0–500 milliseconds, we could set $\eta = 500$ for example, and such a random delay would randomize the timing information of the keystrokes. Such a random delay imposes an overhead of about 250 milliseconds on average. Unfortunately, if the attacker can monitor the same user login many times and compute the average of the latencies of the password sequences, he can reduce the effectiveness of the randomized noise. For example, if the attacker can get the timing information of a user's SSH authentication for 50 times, the noise contributed by the random delay is only about 20–40 milliseconds. So we should not use this method.

A better way to prevent leakage of inter-keystroke timing information is to send traffic at a constant rate of λ packets per second when the link is active. Choosing λ presents a tradeoff between usability and overhead: Increasing λ reduces the dummy traffic but cause longer latency for the user. Assume, for example, that we set $\lambda = 50$ milliseconds. Since the latency between two keystrokes is usually greater than 50 milliseconds and the network delay is already at least in the tens of milliseconds, this may be a reasonable tradeoff between communication overhead and additional delay. In such a scenario, the SSH client would always send a data packet every 50 milliseconds. Assuming 64 byte packets (40 bytes for IP and TCP headers, and 24 bytes for SSH data), the communication overhead is 1280 bytes/second, which can even fit in low-bandwidth connections, such as modem connections. If no real data needs to be sent, the client will send dummy traffic which the remote machine ignores.[6] If the user types multiple keys in a single time period, the keystrokes are buffered and sent together in the next scheduled packet. While this method prevents the eavesdropper from learning timing information about keystrokes typed at the client side, it does not prevent information leakage from the size of response packets from the remote machine. Hence the server side would also need to send response traffic at a constant packet rate similar to the client side.

8 Conclusion

In this paper, we identified several serious security risks in `SSH` due to two weaknesses of `SSH`: First, the transmitted packets are padded only to an eight-byte boundary (if a block cipher is in use), which reveals the approximate size of the original data. Second, in interactive mode, every individual keystroke that a user types is sent to the remote machine in a separate IP packet immediately after the key is pressed (except for some meta keys such `Shift` or `Ctrl`), which leaks the inter-keystroke timings of users' typing. We showed that these two weaknesses reveal a surprising amount of information on passwords and other text typed over `SSH` sessions (about 1 bit of information per character pair in the case of randomly chosen passwords). This suggests that `SSH` is not as secure as commonly believed.

The lessons we learned and the techniques we developed in this paper apply to a general class of protocols that aim to provide secure channels between machines. We show that timing information opens a new set of risks, and we recommend that developers take care when designing these types of protocols.

Acknowledgement

We would like to thank Adrian Perrig for his great help through all phases of the project. We are indebted to Kris Hildrum, Doantam Phan and Robert Johnson for their help in the testing phase. We would also like to thank Eric Xing for discussions on statistical techniques.

[6]If after a certain timeout (e.g., 10λ) there is still no real data to send, the client can consider the current link is inactive and stop sending dummy traffic until it has data to send again. The timeout period provides a tradeoff between security and efficiency.

Finally we would like to thank Nikita Borisov, Monica Chew, Kris Hildrum, Robert Johnson, and Solar Designer for their helpful comments on the paper.

References

[Bel93] Steven M. Bellovin. Packets found on an internet. *Computer Communications Review*, 23(3):26–31, July 1993.

[BSH90] S. Bleha, C. Slivinksy, and B. Hussein. Computer-access security systems using keystrokes dynamics. In *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-12*, volume 12, December 1990.

[CB94] William R. Cheswick and Steven M. Bellovin. *Firewalls and Internet Security – Repelling the Wily Hacker*. Professional Computing Series. Addison-Wesley, 1994. ISBN 0-201-63357-4.

[DS01] Solar Designer and Dug Song. Passive analysis of SSH (secure shell) traffic. Openwall advisory OW-003, March 2001.

[GLPS80] R. Gaines, W. Lisowski, S. Press, and N. Shapiro. Authentication by keystroke timing: Some preliminary results. Technical Report Rand report R-256-NSF, Rand corporation, 1980.

[GS96] Simson Garfinkel and Gene Spafford. *Practical UNIX & Internet Security*. O'Reilly & Associates, 1996.

[JG90] Rick Joyce and Gopal Gupta. Identity authentication based on keystroke latencies. *Communications of the ACM*, 33(2):168 – 176, February 1990.

[Koc95] P. Kocher. Cryptanalysis of Diffie-Hellman, RSA, DSS, and other cryptosystems using timing attacks. In *Advances in cryptology, CRYPTO '95*, pages 171–183. Springer-Verlag, 1995.

[LW88] G. Leggett and J. Williams. Verifying identity via keystroke characteristics. *International Journal of Man-Machine Studies*, 28(1):67–76, 1988.

[LWU89] G. Leggett, J. Williams, and D. Umphress. Verification of user identity via keystroke characteristics. *Human Factors in Management Information Systems*, 1989.

[MR97] Fabian Monrose and Avi Rubin. Authentication via keystroke dynamics. In *Proceedings of the 4th ACM Conference on Computer and Communications Security*, pages 48–56, April 1997.

[MRW99] F. Monrose, M. K. Reiter, and S. Wetzel. Password hardening based on keystroke dynamics. In *Proceedings of the 6th ACM Conference on Computer and Communications Security*, November 1999.

[NBS77] National Bureau of Standards. Specification for the Data Encryption Standard. Federal Information Processing Standards Publication 46 (FIPS PUB 46), January 1977.

[NIS99] U. S. National Institute of Standards and Technology (NIST). Data Encryption Standard (DES). Draft Federal Information Processing Standards Publication 46-3 (FIPS PUB 46-3), January 1999.

[RLCM98] J. A. Robinson, V. M. Liang, J. A. Chambers, and C. L. MacKenzie. Computer user verification using login string keystroke dynamics. *IEEE Transactions on System, Man, and Cybernetics*, 28(2), 1998.

[RN95] Stuart Russell and Peter Norvig. *Artificial Intelligence, A modern approach.* Prentice Hall, 1995.

[Sha50] Claude E. Shannon. Prediction and Entropy of Printed English. Bell Sys. Tech. J (3), 1950.

[SSL01] IETF Secure Shell Working Group (SECSH). `http://www.ietf.org/html.charters/secsh-charter.html`, 2001.

[Tro98] Jonathan Trostle. Timing attacks against trusted path. In *IEEE Symposium on Security and Privacy*, 1998.

[UW85] D. Umphress and J. Williams. Identity verification through keyboard characteristics. *International Journal of Man-Machine Studies*, 23(3):263–273, 1985.

[YKS+00a] T. Ylönen, T. Kivinen, M. Saarinen, T. Rinne, and S. Lehtinen. SSH authentication protocol. Internet Draft, Internet Engineering Task Force, May 2000. Work in progress.

[YKS+00b] T. Ylönen, T. Kivinen, M. Saarinen, T. Rinne, and S. Lehtinen. SSH protocol architecture. Internet Draft, Internet Engineering Task Force, May 2000. Work in progress.

[Ylö96] Tatu Ylönen. SSH – Secure Login Connections over the Internet. In *Sixth USENIX Security Symposium*, San Jose, California, July 1996.

[Zim95] Philip R. Zimmermann. *The Official PGP User's Guide.* MIT Press, Cambridge, MA, USA, 1995. ISBN 0-262-74017-6.

[ZP00a] Yin Zhang and Vern Paxson. Detecting backdoors. In *Proc. of 9th USENIX Security Symposium*, August 2000.

[ZP00b] Yin Zhang and Vern Paxson. Detecting stepping stones. In *Proc. of 9th USENIX Security Symposium*, August 2000.

A The *n*-Viterbi Algorithm

The Viterbi algorithm is widely used in solving HMM problems. Given an observation $(y_1,\dots,y_T)$ of a HMM, the Viterbi algorithm inductively computes the most likely sequence $(q_1,q_2,\dots,q_t)$ that generated the observation for each $t = 1,2,\dots,T$. Let $S(q_t)$ be the most likely sequence at time t that ends with state q_t, with corresponding posterior probability $V(q_t)$. The Viterbi algorithm starts with

$$S(q_1) = q_1 \qquad \text{and} \qquad V(q_1) = \Pr[q_1|y_1],$$

and computes

$$V(q_t) = \max_{q_{t-1}} \Pr[y_t|q_t]\Pr[q_t|q_{t-1}]V(q_{t-1})$$

Then we let q_{t-1} be the state that maximizes the above expression and define $S(q_t)$ to be $S(q_{t-1})|q_t$. The final result of the Viterbi algorithm returns the most likely sequence of a given sequence of observations.

We extend the Viterbi algorithm to the *n*-Viterbi algorithm, which returns the *n* most likely sequences given a sequence of observations. Figure 12 shows a diagram of the *n*-Viterbi algorithm. At each time slice *t*, we associate a list with each possible state node that keeps track of the *n* most likely sequences that lead to the state at that time slice.

Let $S^n(q_t)$ denote the set of the *n* most likely sequences ending with state q_t at time *t*, with corresponding posterior probabilities $V^n(q_t)$. At time $t = 1$, we initialize the *n*-Viterbi algorithm in the same way as the Viterbi algorithm,

$$S^n(q_1) = \{q_1\} \qquad \text{and} \qquad V^n(q_1) = \Pr[q_1|y_1].$$

For time *t*, we let

$$V^n(q_t) = \text{nmax} \quad \{\Pr[y_t|q_t]\Pr[q_t|q_{t-1}]v \; : q_{t-1} \in Q, v \in V^n(q_{t-1})\}$$

where nmax denotes the set of the *n* largest values. We let $S^n(q_t)$ be the set *n* highest-probability sequences corresponding to the choice of $V^n(q_t)$ above.

Except for the first and the second step, at each time slice, for each possible state, we need to go through $n \cdot |Q|$ possibilities and compute the *n* most likely sequences that lead to that state at that time slice. Hence the complexity of *n*-Viterbi algorithm is $O(n|Q|^2T)$.

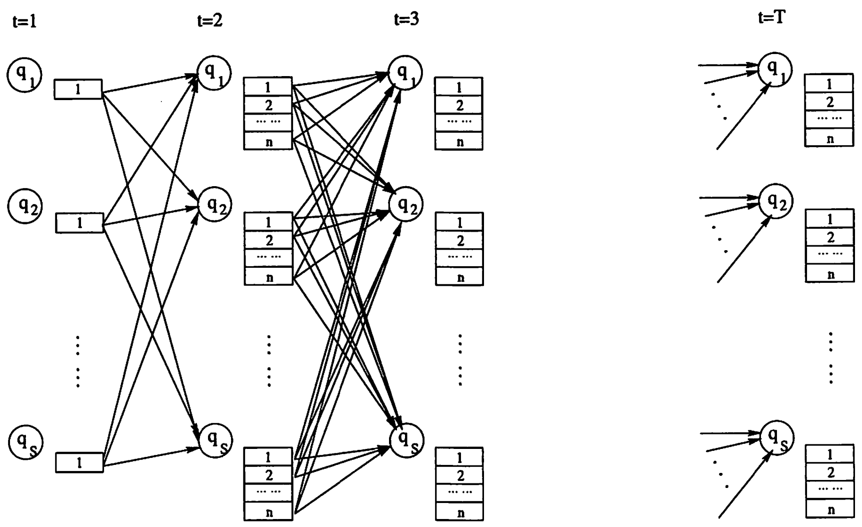


Figure 12: A pictorial representation of the n-Viterbi Algorithm. Each vertical slice represents a time step, and each node represents a possible state at a particular time slice. The list associated with each node stores the n most likely sequences ending with that state up to that time slice.

Reading Between the Lines: Lessons from the SDMI Challenge

Scott A. Craver, Min Wu, Bede Liu,
Department of Electrical Engineering, Princeton University

Adam Stubblefield, Ben Swartzlander, Dan S. Wallach,
Department of Computer Science, Rice University

Drew Dean,

Edward W. Felten
Department of Computer Science, Princeton University

Abstract

The Secure Digital Music Initiative is a consortium of parties interested in preventing piracy of digital music, and to this end they are developing architectures for content protection on untrusted platforms. SDMI recently held a challenge to test the strength of four watermarking technologies, and two other security technologies. No documentation explained the implementations of the technologies, and neither watermark embedding nor detecting software was directly accessible to challenge participants. We nevertheless accepted the challenge, and explored the inner workings of the technologies. We report on our results here.

1 Introduction

The Secure Digital Music Initiative (SDMI), a consortium of music-industry companies, is working to develop and standardize technologies that "protect the playing, storing, and distributing of digital music." [4] SDMI has released little information to the public about its technologies.

In September 2000, SDMI announced a "public challenge" in which it invited members of the public to try to break certain data-encoding technologies that SDMI had developed [6]. The challenge offered a valuable window into SDMI, not only into its technologies but also into its plans and goals. We decided to use the challenge to learn as much as we could about SDMI. This paper is the result of our study.[1] Section 2 presents an overview of the SDMI challenge. Section 3 analyzes the watermark challenges. Section 4 analyzes the non-watermark challenges. Finally, we present our conclusions in section 5.

2 The SDMI Challenge

The SDMI challenge extended over roughly a three-week period, from September 15, 2000 until October 8, 2000. The challenge actually consisted of six sub-challenges, named with the letters A through F, each involving a different technology developed by SDMI's members. We believe these challenges correspond to submissions to the SDMI's Call for Proposals for Phase II Screening Technology [5]. According to this proposal, the watermark's purpose is to enforce a usage policy for an audio clip which is compressed or has previously been compressed. That is, an audio clip possessing a watermark may be admitted into an SDMI device, but only if it has not been degraded by compression.

For each challenge, SDMI provided some information about how a technology worked, and then challenged the

[1]The SDMI challenge offered a small cash payment to be shared among everyone who broke one of the technologies according to criteria set by SDMI, and who was willing to sign a confidentiality agreement giving up all rights to discuss their findings. We chose to forgo the payment and retain our right to publish this paper.

public to create an object with a certain property. The exact information provided varied among the challenges, although we note that in all six cases SDMI provided less information than a would-be copyright infringer would have access to in practice.

In addition, the music clips provided by SDMI in connection with the challenge came with a "click-through agreement" that allowed the files to be used only during the three-week challenge period. The limited time allowed for the challenge, and the apparent prohibition on certain follow-up research beyond the three-week period, prevented us from doing more extensive testing, leaving some of our results in an incomplete state. Of course, a would-be copyright infringer presumably would not be deterred by such legal agreements.

2.1 Watermark Challenges

Four of the challenges (A, B, C, and F), involved watermarking technologies, in which subtle modifications are made to an audio file to encode information without perceptible change in how the file sounds. Watermarks can be either *robust* or *fragile:* robust watermarks are designed to survive common transformations like digital-to-audio conversion, compression and decompression, and the addition of small amounts of noise to the file; whereas fragile watermarks do not survive such transformations, and are used to indicate modification of the file.

For each of the four watermark challenges, SDMI provided three files:

- *File 1*: an unwatermarked song;
- *File 2*: File 1, with a watermark added; and
- *File 3*: another watermarked song.

The challenge was to produce a file that sounded just like File 3 but did not have a watermark — in other words, to render the watermark undetectable.

SDMI provided an on-line "oracle" for each challenge. Entrants could send a file to the oracle, and the oracle would inform them if their submission satisfied the challenge, that is, if it contained no detectable watermark while still sounding like File 3. Entrants were given no information about how watermark information was stored in the file or how the oracle detected watermarks, beyond the information that could be deduced from inspection of the three provided files and the oracle's output.

2.2 Challenges D and E

Challenge D concerned a technology designed to prevent a song from being separated from the album in which it was issued. Normally, every Compact Disc contains a table of contents, indicating the offsets and lengths of each audio track, followed by the audio data itself. Challenge D adds an "authenticator" track (approximately 50ms of very quiet audio), derived somehow from the table of contents. The authenticator is supposed to be difficult to compute for an arbitrary CD. Challenge D is discussed in more detail in Section 4.1.

Challenge E involved a technology similar to D, but one which would be immune to the obvious attack on technology D, in which one compiled an unauthorized CD with the same table of contents as an authorized one for which the authenticator track is known. Unfortunately, this challenge was constructed in a way that made it impossible to even start analyzing the technology. SDMI provided an oracle for this challenge, but unfortunately provided no music samples of any kind, so there was no way to determine what the oracle might be testing for.

Given these facts, we decided not to analyze Challenge E. It is discussed briefly in Section 4.2.

3 The Watermarking Schemes

In this section, we describe our attack(s) on each of the four watermark challenges (A,B,C,F). Our success was confirmed by emails received from SDMI's oracles.

Figure 1 provides an overview of the challenge goal. As mentioned earlier, there are three audio files per watermark challenge: an original and watermarked version of one clip, and then a watermarked version of a second clip, from which the mark is to be removed. All clips were 2 minutes long, sampled at 44.1kHz with 16-bit precision.

The reader should note one serious question regarding this challenge arrangement. The challenge is to render a robust mark undetectable, while these technologies appear to be Phase II watermark screening tech-

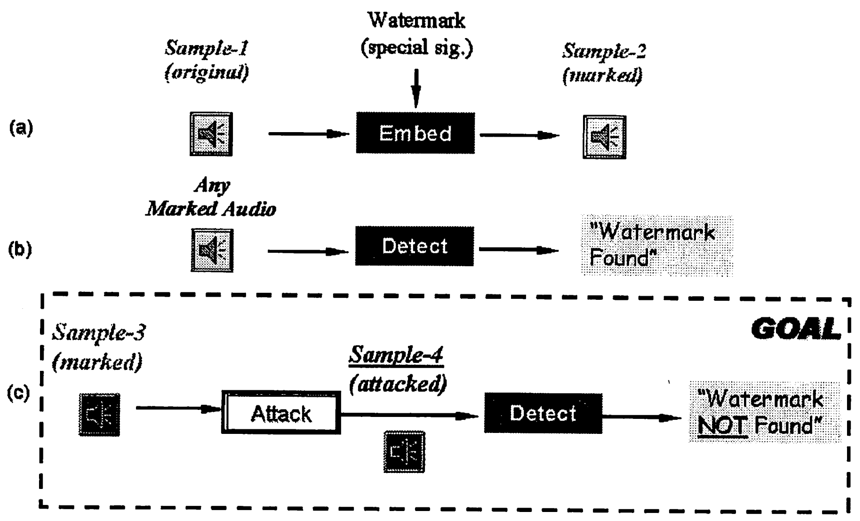


Figure 1: The SDMI watermark attack problem. For each of the four watermark challenges, Sample-1, sample-2, and sample-3 are provided by SDMI. Sample-4 is generated by participants in the challenge and submitted to SDMI oracle for testing.

nologies [5]. As we mentioned above, a Phase II screen is intended to reject audio clips if they have been compressed, and presumably compression degrades a fragile component of the watermark. An attacker need not remove the robust watermark to foil the Phase II screen, but alternatively could repair the modified fragile component in compressed audio. This attack was not available under the challenge setup.

Our analysis of the watermarking schemes uses standard signal processing methods. The text below presumes the reader has a basic understanding of signal processing. Readers without such a background might want to consult a source such as the textbook by Steiglitz [7].

3.1 Attack and Analysis of Technology A

A reasonable first step in analyzing watermarked content with original, unmarked samples is differencing the original and marked versions in some way. Initially, we used sample-by-sample differences in order to determine roughly what kinds of watermarking methods were taking place. Unfortunately, technology A involved a slowly varying phase distortion which masked any other cues in a sample-by-sample difference. We ultimately decided this distortion was a pre-processing separate from the watermark, in part because undoing the distortion alone did not foil the oracle.

The phase distortion nevertheless led us to attempt an attack in which both the phase and magnitude change between sample 1 and sample 2 is applied to sample 3. The attack was based on this code, where FFT computes the Fast Fourier Transform (FFT), and IFFT computes the inverse FFT.

```
while(framesLeftInSong()){
  Y = FFT(nextFrame(markedMusicFile));
  X = FFT(nextFrame(unmarkedMusicFile));
  H = elementwiseDivide(Y,X);
  Z = FFT(nextFrame(otherMarkedFile));
  R = elementwiseDivide(Z,H);
  outputFrame(IFFT(R));
}
```

This attack was confirmed by SDMI's oracle as successful, and illustrates the general attack approach of imposing the difference in an original-watermark pair upon another media clip. Here, the "difference" is taken in the frequency domain rather than the time domain; we suspected a watermark based on frequency-domain modification because of the apparent presence of phase distortion in the watermarking process. Note that this attack did not require much information about the watermarking scheme itself, and conversely did not provide much extra insight into its workings.

A next step, then, is to compute the frequency reponse $H(\omega) = W(\omega)/O(\omega)$ of the watermarking process for segments of audio, where W is the (frequency-domain)

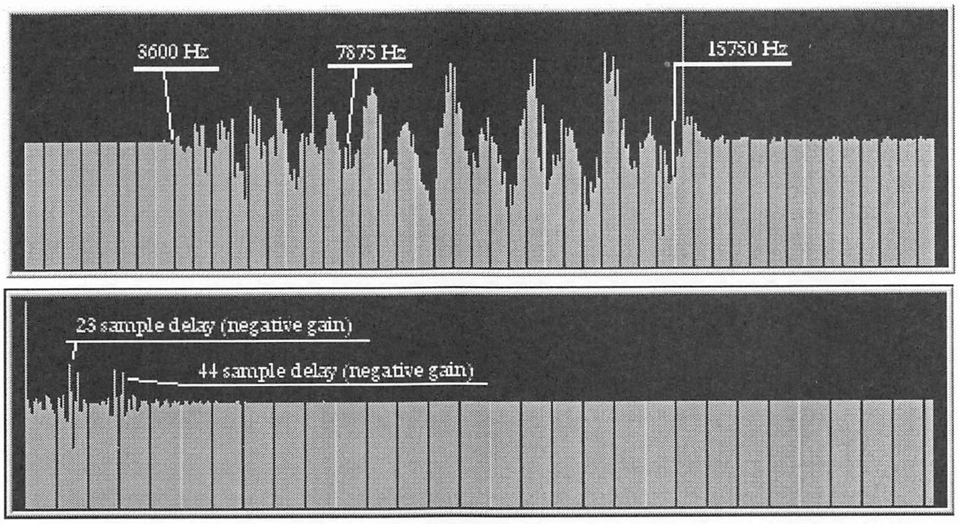


Figure 2: A short-term complex echo. Above, the frequency response between the watermarked and original music, taken over 1/50 second, showing a sinusoidal ripple between 8 and 16 KHz. Below, the corresponding impulse response. The sinusoidal pattern in the frequency domain corresponds to a pair of echoes in the time domain.

watermarked signal and O is the (frequency-domain) original signal, and to observe both $|H(\omega)|$ and the corresponding impulse response $h(t)$. If the watermark is based on some kind of linear filter, whose properties change slowly enough relative to the size of a frame of samples, then this approach is ideal.

Figure 2 illustrates one frequency response and impulse response about 0.3 seconds into the music. These responses are based on FFTs of 882 samples, or one fiftieth of a second of music. As can be clearly seen, a pair of sinusoidal ripples are present within a certain frequency band, approximately 8-16Khz. An echo in the time domain manifests itself as a ripple in the frequency domain, so a sum of sinusoids in the frequency domain suggests the presense of multiple echoes. The corresponding impulse response $h(t)$ confirms this. This pattern of ripples changes quite rapidly from frame to frame.

Thus, we had reason to suspect a complex echo hiding system, involving multiple time-varying echoes. It was at this point that we considered a patent search, knowing enough about the data hiding method that we could look for specific search terms, and we were pleased to discover that this particular scheme appears to be listed as an alternative embodiment in US patent number 5,940,135, awarded to Aris corporation, now part of Verance [3]. This provided us with little more detail than we had already discovered, but confirmed that we were on the right track, as well as providing the probable identity of the company which developed the scheme. It also spurred no small amount of discussion of the validity of Kerckhoffs's criterion, the driving principle in security that one must not rely upon the obscurity of an algorithm. This is, surely, doubly true when the algorithm is patented.

The most useful technical detail provided by the patent was that the "delay hopping" pattern was likely discrete rather than continuous, allowing us to search for appropriate frame sizes during which the echo parameters were constant. Data collection from the first second of audio showed a frame size of approximately 882 samples, or 1/50 second. We also observed that the mark did not begin until 10 frames after the start of the music, and that activity also existed in a band of lower frequency, approximately 4-8 Khz. This could be the same echo obscured by other operations, or could be a second band used for another component in the watermarking scheme. A very clear ripple in this band, indicating a single echo with a delay of about 34 samples, appears shortly before the main echo-hopping pattern begins.

The next step in our analysis was the determination of the delay hopping pattern used in the watermarking method, as this appeared to be the "secret key" of the data embedding scheme. It is reasonable to suspect that the pattern repeats itself in short order, since a watermark detector should be able to find a mark in a subclip of music, without any assistance initially aligning the mark with the detector's hopping pattern. Again, an analysis of the first second revealed a pattern of echo pairs that appeared to repeat every 16 frames, as outlined in figure 3. The delays appear to fall within six general categories, each

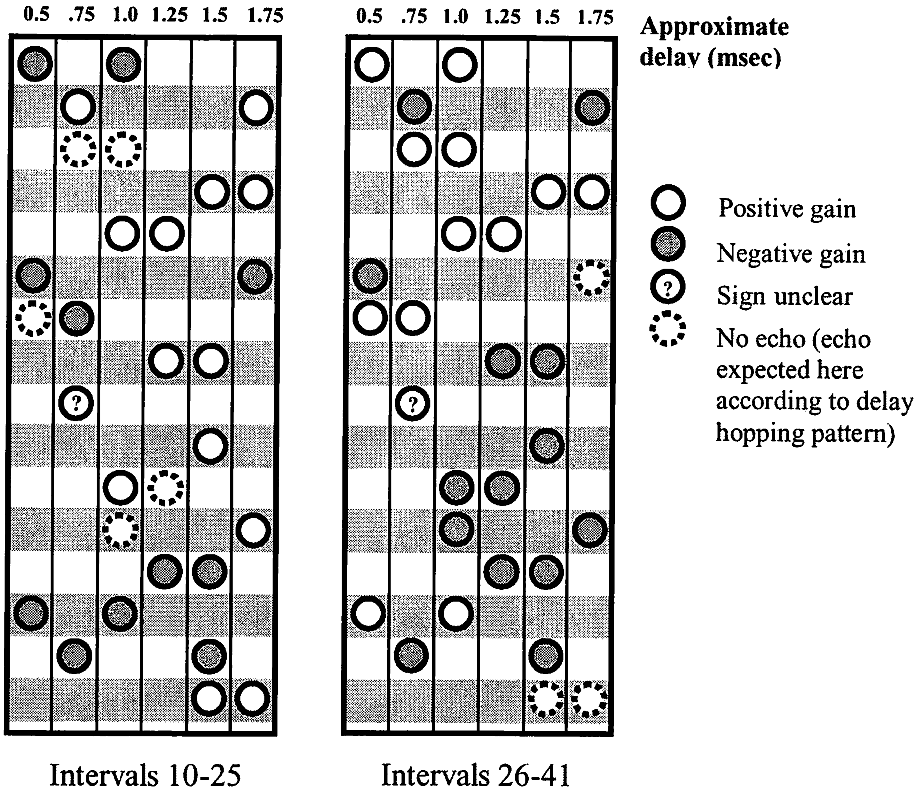


Figure 3: The hypothesized delay hopping pattern of technology A. Here two stretches of 16 frames are illustrated side-by-side, with observed echoes in each frame categorized by six distinct delays: 2, 3, 4, 5, 6 or 7 times 0.00025 sec. Aside from several missing echoes, a pattern appears to repeat every 16 frames. Note also that in each frame the echo gain is the same for both echoes.

delay approximately a multiple of 1/4 millisecond. The exact values of the delays vary slightly, but this could be the result of the phase distortion present in the music.

The reader will also note that in apparently two frames there is only one echo. We found two possible explanations for this. First, if we made two independent random choices from six possible echo delays for each frame, then we would expect the two random choices to coincide (i.e., to be equal to each other) about one-sixth of the time; the observed ratio of two coincidences in sixteen frames is consistent with this hypothesis. Second, our detector appears to have missed some echoes, which show up in one of the two sides of the figure but not the other. (These are depicted as dotted circles on the side where they appear to have been missed.) If the detector happened to miss the same echo in *both* sides of the figure, this would lead to a frame that appeared to have only one echo. Again, the frequency of missing echoes is roughly consistent with this hypothesis. Of course, it may be that some other explanation is correct.

Next, there is the issue of the actual encoded bits. Further work shows the sign of the echo gain does not repeat with the delay-hopping pattern, and so is likely at least part of an embedded message. Extracting such data without the help of an original can be problematic, although the patent, of course, outlines numerous detector structures which can be used to this end. We developed several tools for cepstral analysis to assist us in the process. See [2] for in introduction to cepstral analysis; Anderson and Petitcolas [1] illustrate its use in attacks on echo hiding watermark systems.

With a rapidly changing delay, normal cepstral analysis does not seem a good choice. However, if we know that the same echo is likely to occur at multiples of 16/50 of a second, we can improve detector capability by combining the information of multiple liftered[2] log spectra.

[2]In accordance with the flopped vocabulary used with cepstral analysis, "liftering" refers to the process of filtering data in the frequency domain rather than the time domain. Similarly, "quefrencies" are frequencies of ripples which occur in the frequency domain rather than the time domain.

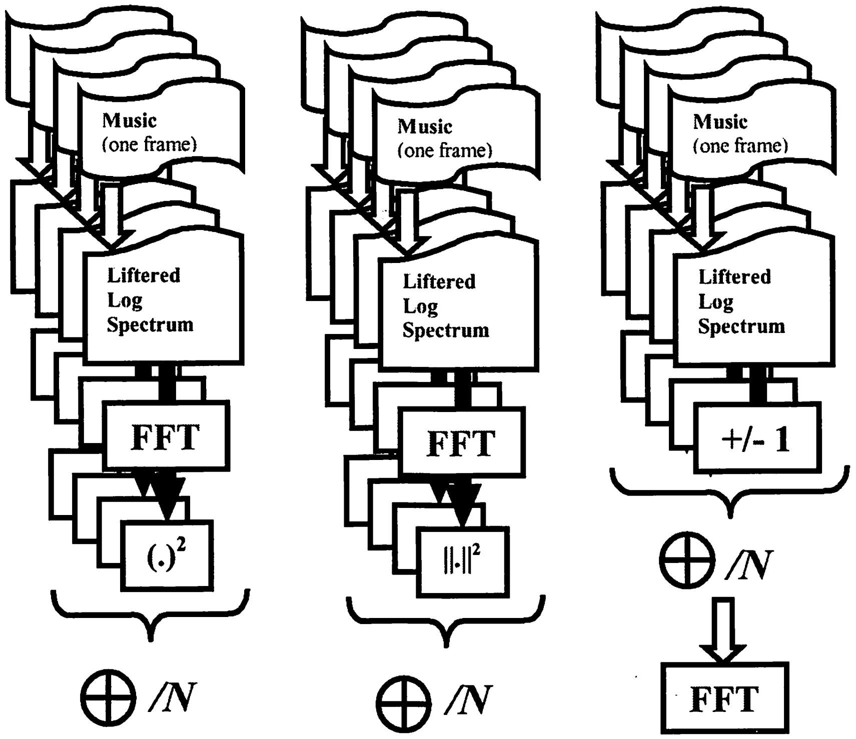


Figure 4: Three cepstral detector structures. In each case we have a collection of distinct frames, each believed to possess echoes of the same delay. The first two compute cepstral data for each frame, and sum their squares (or squared magnitudes) to constructively combine the echo signal in all frames. The third structure illustrates a method for testing a hypothesized pattern of positive and negative gains, possibly useful for brute-forcing or testing for the presence of a known "ciphertext."

Three detector structures are shown in figure 4. In all three, a collection of frames are selected for which the echo delays are believed to be the same. For each, the liftered log of an FFT or Power Spectral Distribution (PSD) of the frame is taken. In the first two structures, we compute a cepstrum for each frame, then either average their squared magnitudes, or simply their squares, in hopes that a spike of the appropriate quefrency will be clear in the combination. The motivation for merely squaring the spectral coefficients comes from the observation that a spike due to an echo will either possess a phase of ϕ or $\phi + \pi$ for some value ϕ. Squaring without taking magnitudes can cause the echo phases to reinforce, whilst still permitting other elements to combine destructively.

In the final structure, one cepstrum is taken using a guess of the gain sign for each suspect frame. With the correct guess, the ripple should be strongest, resulting in the largest spike from the cepstral detector. Figure 5 shows the output of this detector on several sets of suspect frames. While this requires an exponential amount of work for a given number of frames, it has a different intended purpose: this is a brute-forcing tool, a utility for determining the most probable among a set of suspected short strings of gain signs as an aid to extracting possible ciphertext values.

Finally, there is the issue of what this embedded watermark means. Again, we are uncertain about a possible signalling band below 8KHz. This could be a robust mark, signalling presence of a fragile mark of echoes between 8 and 16 KHz. The 8-16KHz band does seem like an unusual place to hide robust data, since compressors often reduce the information in this band greatly. If the signal does indeed extend further down, the 8-16KHz band could very easily be hidden information whose degradation is used to determine if music has already been compressed.

Of course, knowledge of *either* the robust or fragile component of the mark is enough for an attacker to defeat the scheme, because one can either remove the robust mark, or repair or reinstate the fragile mark after compression

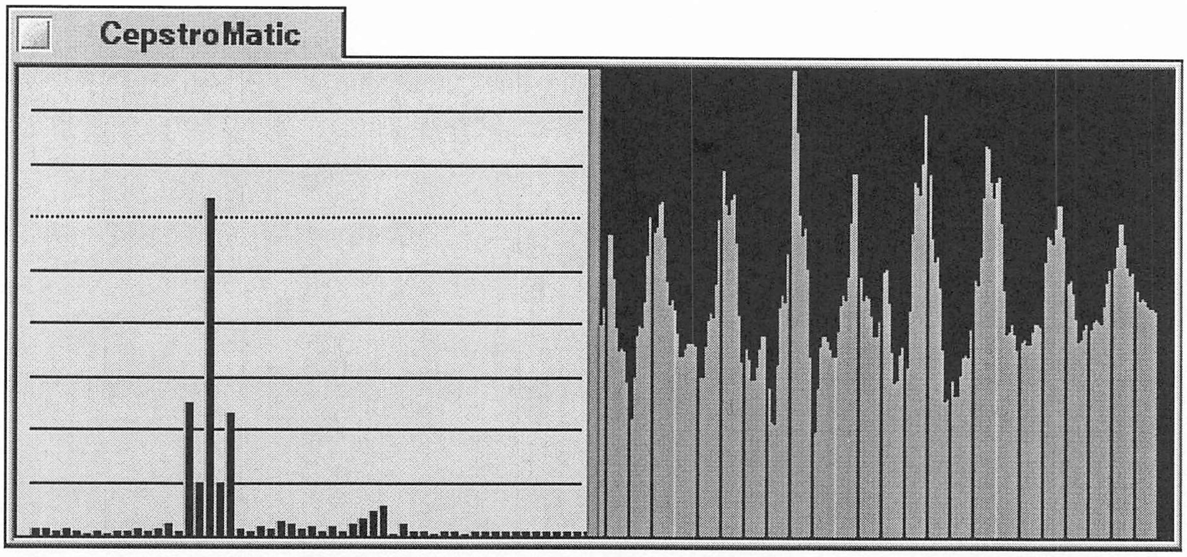


Figure 5: Detection of an echo. A screenshot of our CepstroMatic utility shows a combination of 4 separate frames of music, each a fiftieth of a second long, in which the same echo delay was believed to exist. Their combination shows a very clear ripple on the right, corresponding to a clear cepstral spike on the left. This is a single echo at a delay of 33 samples, the delay suggested for these intervalus by the hypothesized delay-hopping pattern.

has damaged it. As mentioned earlier, this possible attack of repairing the fragile component appears to have been ruled out by the nature of the SDMI challenge oracles. One must wait and see if real-world attackers will attempt such an approach, or resort to brute force methods or oracle attacks to remove the robust component.

3.2 Attack on Challenge B

For Challenge B, we analyzed the matching unwatermarked and watermarked samples using a short-time FFT. Shown in Fig. 6 are the two FFT magnitudes for 1000 samples at 98.67 sec. Also shown is the difference of the two magnitudes. A spectrum notch around 2800Hz is observed for some segments of the watermarked sample and another notch around 3500Hz is observed for some other segments of the watermarked sample. Similar notches are observed in the other watermarked sample. The attack fills in those notches with random but bounded coefficient values. We also submitted a variation of this attack involving different parameters for notch description. Both attacks were confirmed by the SDMI oracle as successful.

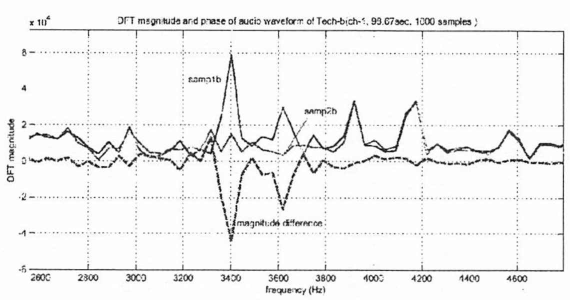


Figure 6: Challenge B: FFT magnitudes of the matching watermarked (samp1b) and unwatermarked (samp2b) files and their difference for 1000 samples at 98.67 sec.

3.3 Attacks on Challenge C

By taking the difference between the matching watermarked and unwatermarked samples for Challenge C, we observed bursts of narrowband signal, as shown in Fig. 7. These narrow band bursts appear to be centered around 1350 Hz, suggesting that the detector is looking for something to be present at 1350 Hz. We applied two different attacks to Challenge C. In the first attack, we shifted the pitch of the audio by about a quartertone to move the bursts away from 1350 Hz. In the second attack, we passed the signal through a bandstop filter centered around 1350Hz. Both submissions were confirmed

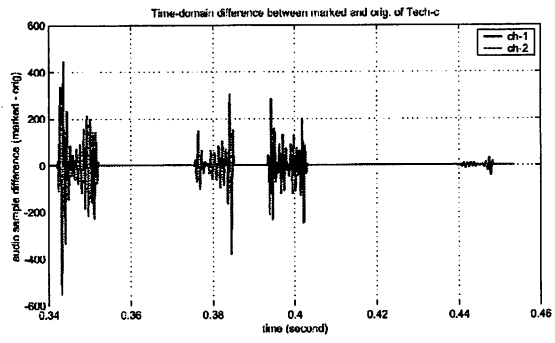


Figure 7: Challenge C: Waveform of the difference between the matching watermarked and unwatermarked files.

by SDMI oracle as successful. In addition, the perceptual quality of both attacks passed the "golden ear" testing conducted by SDMI after the 3-week challenge.

3.4 Attack on Challenge F

For Challenge F, we warped the time axis, by inserting a periodically varying delay. Essentially, we computed $y(t) = x(t + f(t))$, in which x is the original audio signal and f is a slowly-varying sinusoidal function. In our first successful attack, $f(t) = 3(1 - cos(0.602\pi t))^2/19600$, where t is measured in seconds. This has a period of approximately 3.32 seconds, and distorts the music by a maximum of 27 samples, or about 0.6 milliseconds. The attack is implemented by the code shown in Figure 8.

The delay function comes from our study of Technology A, and was in fact initially intended to undo the phase distortion applied by Technology A. Therefore, the perceptual quality of our attacked audio is expected to be better than or comparable to that of the audio watermarked by Technology A. We also submitted variations of this attack involving different warping parameters and different delay functions. They were confirmed by the SDMI oracle as successful.

```
int main(int argc, char* argv[]){
  int k, numSamples;
  int left,right,oldleft,oldright;
  int cumulative = 0;
  double delay;

  numSamples = getNumSamples();
  for(k=0; k<numSamples; ++k){
    readsample(&left, &right);
    delay = 1.0-cos(0.602*PI*k/44100);
    delay = 6.75*delay*delay;
    if((int)delay < cumulative){
      cumulative--;
      writesample(oldleft, oldright);
      writesample(left, right); break;
    }else if((int)delay > cumulative){
      cumulative++;
    }else{
      writesample(left, right);
    }
    oldleft = left; oldright = right;
  }
  return 0;
}
```

Figure 8: Code for the time-axis warping attack on Challenge F.

4 The Non-Watermark Technologies

The SDMI challenge contained two "non-watermark" technologies. Together, they appear to be intended to prevent the creation of "mix" CDs, where a consumer might compile audio files from various locations to a writable CD. This would be enforced by universally embedding SMDI logic into consumer audio CD players.

4.1 Technology D

According to SDMI, Technology D was designed to require "the presence of a CD in order to 'rip' or extract a song for SDMI purposes." The technology aimed to accomplish this by adding a 53.3 ms audio track (four blocks of CD audio), which we will refer to as the **authenticator**, to each CD. The authenticator, combined with the CD's table of contents (TOC), would allow an SDMI device to recognize SDMI compliant CDs. For the challenge, SDMI provided 100 different "correct" TOC-authenticator pairs as well as 20 "rogue tracks". A rogue track is a track length that does not match any of the track lengths in the 100 provided TOCs. The goal of the challenge was to submit to the SDMI oracle a correct authenticator for a TOC that contained at least one of the rogue tracks.

The oracle for Technology D allowed several different query types. In the first type, an SDMI provided TOC-authenticator combination is submitted so that a user can "understand and verify the Oracle." According to SDMI, the result of this query should either be "admit" for a correct pair or "reject" for an incorrect pair. When we attempted this test with an SDMI-provided pair, the oracle responded that the submission was "invalid." After verifying that we had indeed submitted a correct pair, we attempted several other submissions using different TOC-authenticator pairs as well as different browsers and operating systems[3]. We also submitted some pairs that the oracle should have rejected; these submissions were also declared "invalid." Though we alerted SDMI to this problem during the challenge, the oracle was never repaired. For this reason, our analysis of Technology D is incomplete and we lack definitive proof that it is correct. That having been said, we think that what we learned about this technology, even without the benefit of a correctly functioning oracle, is interesting.

4.1.1 Analyzing the Signal

Upon examination of the authenticator audio files, we discovered several patterns. First, the left and right channels contain the same information. The two channels differ by a "noise vector" **u**, which is a vector of small integer values that range from -8 and 8. Since the magnitude of the noise is so small, the noise vector does not significantly affect the frequency characteristics of the signal. The noise values appear to be random, but the noise vector is the same for each of the 100 provided authenticator files. In other other words, in any authenticator file, the difference between the left and right channels of the ith sample is a constant fixed value $\mathbf{u}[i]$. This implies that the noise vector **u** does not encode any TOC-specific information.

Second, the signal repeats with a period of 1024 samples. Because the full signal is 2352 samples long, the block repeats approximately 1.3 times. Similarly to the left and right channels of the signal, the first two iterations of the repeating signal differ by a constant noise vector **v**. The difference between the ith sample of the first iteration and the ith sample of the second iteration differ by a small (and apparently random) integer value $\mathbf{v}[i]$ ranging from -15 to 15. In addition, **v** is the same for each of the provided authenticator files, so **v** does not encode any TOC-specific information.

[3]Specifically, Netscape Navigator and Mozilla under Linux, Netscape Navigator under Windows NT, and Internet Explorer under Windows 98 and 2000.

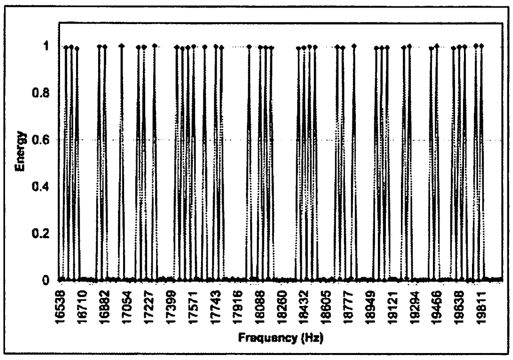


Figure 9: Individual Bits From a Technology D Authenticator

Third, the first 100 samples and last 100 samples of the full signal are faded in and faded out, respectively. The fade-in and fade-out are meaningless, however, because they simply destroy data that is repeated in the middle of the file. We conjecture that this fade-in and fade-out are included so that the audio signal does not sound offensive to a human ear.

4.1.2 Extracting the Data

Frequency analysis on the 1024 sample block shows that almost all of the signal energy is concentrated in the 16–20kHz range. We believe this range was chosen because these frequencies are less audible to the human ear. Closer examination shows that this 16–20kHz range is divided up into 80 discrete bins, each of which appears to carry one bit of information. As shown in Figure 9, these bits can be manually counted by a human using a graph of the magnitude of signal in the frequency domain.

Close inspection and pattern matching on these 80 bits of information reveals that there are only sixteen bits of information repeated five times using different permutations. Using the sixteen letters A-P to symbolize the sixteen bits, these five permutations are described in Figure 10.

Because of the malfunctioning oracle, we were unable to determine the function used to map TOCs to authenticators, but given an actual SDMI device, it would be trivial to brute force all 2^{16} possibilities. Likewise, without the oracle, we could not determine if there was any other signal present in the authenticator (*e.g.*, in the phase of

ABCDEFGHIJKLMNOP
OMILANHGPBDCKJFE
PKINHODFMJBCAGLE
FCKLGMEPNOADJBHI
PMGHLECAKDONIFJB

Figure 10: The encoding of the 16 bits of data in Technology D

the frequency components with nonzero magnitude).

For the moment, let us assume that the hash function used in Technology D has only 16 bits of output. Given the number of distinct CDs available, an attacker should be able to acquire all, or almost all, of the authenticators. We note that at 9 kilobytes each, a collection of 65,536 files would fit nicely on a single CD. Many people have CD collections of 300+ discs, which by the birthday paradox makes it more likely than not that there is a hash collision among their own collection.

Our results indicated that the hash function used in Technology D could be weak or may have less than 16 bits of output. In the 100 authenticator samples provided in the Technology D challenge, there were two pairs of 16-bit hash collisions. We will not step through the derivation here, but for $n < X$, the probability of two or more collisions occurring in n samples of X equally likely possibilities is:

$$1 \quad \left(\prod_{i=1}^{n-1} \frac{X+1-i}{X}\right)\left(1 \quad \frac{n^2-3n+2}{2X}\right)$$

If the 16-bit hash function output has 16 bits of entropy, the probability of 2 collisions occurring in $n = 100$ samples of $X = 2^{16}$ possibilities is 0.00254 (by the above equation). If $X \approx 2^{11.5}$, the chances of two collisions occurring is about even. This suggests that either 4 bits of the 16-bit hash output may be outputs of functions of the other 12 bits or the hash function used to generate the 16-bit signature is weak. It is also possible that the challenge designers purposefully selected TOCs that yield collisions. The designers could gauge the progress of the contestants by observing whether anyone submits authenticator A with TOC B to the oracle, where authenticator A is equal to authenticator B. Besides the relatively large number of collisions in the provided authenticators, it appears that there are no strong biases in the authenticator bits such as significantly more or fewer 1's than 0's.

4.2 Technology E

Technology E is designed to fix a specific bug in Technology D: the TOC only mentions the *length* of each song but says nothing about the contents of that song. Accordingly, an attacker wishing to produce a mix CD would only need to find a TOC approximately the same as that of the desired mix CD, then copy the TOC and authenticator from that CD onto the mix CD. If the TOC does not perfectly match the CD, the track skipping functionality will still work but will only get "close" to track boundaries rather than reaching them precisely. Likewise, if a TOC specified a track length longer than the track we wished to put there, we could pad the track with digital silence (or properly SDMI-watermarked silence, copied from another valid track). Regardless, a mix CD played from start to end would work perfectly. Technology E is designed to counter this attack, using the audio data itself as part of the authentication process.

The Technology E challenge presented insufficient information to be properly studied. Rather than being given the original audio tracks (from which we might study the unspecified watermarking scheme), we were instead given the tables of contents for 1000 CDs and a simple scripting language to specify a concatenation of music clips from any of these CDs. The oracle would process one of these scripts and then state whether the resulting CD would be rejected.

While we could have mounted a detailed statistical analysis, submitting hundreds or thousands of queries to the oracle, we believe the challenge was fundamentally flawed. In practice, given a functioning SDMI device and actual SDMI-protected content, we could study the audio tracks in detail and determine the structure of the watermarking scheme.

We later received hints that Technology E may have been susceptible to attack despite the very limited information we were given. If true, this suprising assertion may in itself convey information about how Technology E works. Unfortunately, we did not receive these hints until after the challenge was over, when we no longer had access to the oracle to study the matter further.

5 Conclusion

In this paper, we have presented an analysis of the technology challenges issued by the Secure Digital Music

Initiative. Each technology challenge described a specific goal (*e.g.*, render undetectable a watermark from an audio track) and offered a Web-based oracle that would confirm whether the challenge was successfully defeated. We have defeated all four of their audio watermarking technologies, and have studied and analyzed their "non-watermarking" technologies to the best of our abilities given the lack of information available to us and given a broken oracle in one case.

Some debate remains as to whether our attacks damaged the audio beyond standards measured by "golden ear" human listeners. Given a sufficient body of SDMI-protected content using the watermark schemes presented here, we are confident we could refine our attacks to introduce distortion no worse than the watermarks themselves introduce to the the audio. Likewise, debate remains on whether we have truly defeated technologies D and E. Given a functioning implementation of these technologies, we are confident we can defeat them.

References

[1] ANDERSON, R. J., AND PETITCOLAS, F. A. P. On the limits of steganography. *IEEE Journal of Selected Areas in Communications 16*, 4 (May 1998), 474–481.

[2] BOGERT, R. P., HEALY, M. J., AND TUKEY, J. W. The quefrency alanysis of time series for echoes: Cepstrum, pseudo-autocovariance, cross-ceptsrum and saphe-cracking. In *Proceedings of the Symposium on Time Series Analysis* (Brown University, June 1962), pp. 209–243.

[3] PETROVIC, R., WINOGRAD, J. M., JEMILI, K., AND METOIS, E. Apparatus and method for encoding and decoding information in analog signals, Aug. 1999. US Patent No. 5,940,135.

[4] SECURE DIGITAL MUSIC INITIATIVE. `http://www.sdmi.org`.

[5] SECURE DIGITAL MUSIC INITIATIVE. *Call for Proposals for Phase II Screening Technology, Version 1.0*, Feb. 2000. `http://www.sdmi.org/download/FRWG00022401-Ph2_CFPv1.0.PDF`.

[6] SECURE DIGITAL MUSIC INITIATIVE. SDMI public challenge, Sept. 2000. `http://www.hacksdmi.org`.

[7] STEIGLITZ, K. *A Digital Signal Processing Primer: with Applications to Digital Audio and Computer Music.* Addison Wesley, 1996.

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

SAGE, the System Administrators Guild

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX.

Member Benefits:

- Free subscription to *;login:*, the Association's magazine, published eight times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Java, Tcl/Tk, and Open Source, book and software reviews, summaries of sessions at USENIX conferences, and Standards Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to *;login:* on the USENIX Web site.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, on the USENIX Web site.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as security, Linux, Internet technologies and systems, operating systems, and Windows—as many as twelve technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.
- Savings on a variety of products, books, software, and periodicals: see *http://www.usenix.org/membership/specialdisc.html* for details.

Supporting Members of the USENIX Association:

Addison-Wesley	Microsoft Research	Sendmail, Inc.
Kit Cosper	Motorola Australia Software Centre	Smart Storage, Inc.
Earthlink Network	New Riders Publishing	Sun Microsystems, Inc.
Edgix	Nimrod AS	Sybase, Inc.
Interhack Corporation	O'Reilly & Associates Inc.	Syntax, Inc.
Interliant	Raytheon Company	Taos: The Sys Admin Company
Lessing & Partner	Sams Publishing	TechTarget.com
Linux Security, Inc.	The SANS Institute	UUNET Technologies, Inc.
Lucent Technologies		

Supporting Members of SAGE:

Certainty Solutions	Mentor Graphics Corp.	Remedy Corporation
Collective Technologies	Microsoft Research	RIPE NCC
Electric Lightwave, Inc.	Motorola Australia Software Centre	Sams Publishing
ESM Services, Inc.	New Riders Publishing	SysAdmin Magazine
Lessing & Partner	O'Reilly & Associates Inc.	Taos: The Sys Admin Company
Linux Security, Inc.	Raytheon Company	Unix Guru Universe

For more information about membership, conferences, or publications,
see *http://www.usenix.org/*
or contact:
USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA
Phone: 510-528-8649 Fax: 510-548-5738 Email: *office@usenix.org*

ISBN 1-880446-07-3